



conference

proceedings

**27th Large Installation
System Administration
Conference (LISA '13)**

***Washington, D.C., USA
November 3–8, 2013***

Proceedings of the 27th Large Installation System Administration Conference

Washington, D.C., USA November 3–8, 2013

Sponsored by



In cooperation with LOPSA

Thanks to Our LISA '13 Sponsors

Gold Sponsors



Silver Sponsors



Bronze Sponsors



Media Sponsors and Industry Partners

- ACM Queue

ADMIN

CiSE

Computer

Distributed Management Task Force (DMTF)

Free Software Magazine

HPCwire

IEEE Pervasive
- IEEE Security & Privacy

IEEE Software

InfoSec News

IT/Dev Connections

IT Professional

Linux Foundation

Linux Journal

Linux Pro Magazine
- LXer

No Starch Press

O'Reilly Media

Open Source Data Center Conference (OSDC)

Server Fault

The Data Center Journal

Userfriendly.org

© 2013 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. Permission is granted to print, primarily for one person's exclusive use, a single copy of these Proceedings. USENIX acknowledges all trademarks herein.

ISBN 978-1-931971-05-8

Thanks to Our USENIX and LISA SIG Supporters

USENIX Patrons

Google InfoSys Microsoft Research NetApp VMware

USENIX Benefactors

Akamai EMC Hewlett-Packard *Linux Journal*
Linux Pro Magazine Puppet Labs

USENIX and LISA SIG Partners

Cambridge Computer Google

USENIX Partners

Meraki Nutanix

USENIX Association

**Proceedings of the
27th Large Installation System
Administration Conference**

**November 3–8, 2013
Washington, D.C.**

Conference Organizers

Program Co-Chairs

Narayan Desai, *Argonne National Laboratory*
Kent Skaar, *VMware, Inc.*

Program Committee

Patrick Cable, *MIT Lincoln Laboratory*
Mike Ciavarella, *Coffee Bean Software Pty Ltd*
Andrew Hume, *AT&T Labs—Research*
Paul Krizak, *Qualcomm*
Adam Leff, *WebMD*
John Looney, *Google, Inc.*
Andrew Lusk, *Amazon Inc.*
Chris McEniry, *Sony*
Tim Nelson, *Worcester Polytechnic Institute*
Marco Nicosia, *Moraga Systems LLC*
Adam Oliner, *University of California, Berkeley*
Carolyn Rowland, *Twinight Enterprises*
Dan Russel, *TED Talks*
Adele Shakal, *Metacloud*
Avleen Vig, *Etsy, Inc.*

Invited Talks Coordinators

Nicole Forsgren Velasquez, *Utah State University*
Cory Lueninghoener, *Los Alamos National Laboratory*

Lightning Talks Coordinator

Lee Damon, *University of Washington*

Workshops Coordinator

Kyrre Begnum, *Oslo and Akershus University College of Applied Sciences*

Guru Is In Coordinator

Chris St. Pierre, *Amplify*

Gurus

Owen DeLong, *Hurricane Electric*
Stephen Frost, *Resonate*
Thomas A. Limoncelli, *Stack Exchange*

David Nalley, *Apache Cloudstack*

Adele Shakal, *Metacloud, Inc.*

Daniel J Walsh, *Red Hat*

George Wilson, *Delphix*

Charles Wimmer, *VertiCloud*

Poster Session Coordinator

Marc Chiarini, *Harvard SEAS*

USENIX Board Liaisons

David N. Blank-Edelman, *Northeastern University*

Carolyn Rowland, *Twinight Enterprises*

Steering Committee

Paul Anderson, *University of Edinburgh*
David N. Blank-Edelman, *Northeastern University*
Mark Burgess, *CFEngine*
Alva Couch, *Tufts University*
Anne Dickison, *USENIX Association*
Æleen Frisch, *Exponential Consulting*
Doug Hughes, *D. E. Shaw Research, LLC*
William LeFebvre, *CSE*
Thomas A. Limoncelli, *Stack Exchange*
Adam Moskowitz
Mario Obejas, *Raytheon*
Carolyn Rowland, *Twinight Enterprises*
Rudi van Drunen, *Xlexit Technology, The Netherlands*

Education Director

Daniel V. Klein, *USENIX Association*

Tutorial Coordinator

Matt Simmons, *Northeastern University*

LISA Lab Hack Space Coordinators

Paul Krizak, *Qualcomm*
Chris McEniry, *Sony*
Adele Shakal, *Metacloud, Inc.*

External Reviewers

David N. Blank-Edelman and Thomas A. Limoncelli

LISA '13:
27th Large Installation System Administration Conference
November 3–8, 2013
Washington, D.C.

Message from the Program Co-Chairs v

Wednesday, November 6

Building Software Environments for Research Computing Clusters1
Mark Howison, Aaron Shen, and Andrew Loomis, *Brown University*

Fixing On-call, or How to Sleep Through the Night7
Matt Provost, *Weta Digital*

Thursday, November 7

Poncho: Enabling Smart Administration of Full Private Clouds17
Scott Devoid and Narayan Desai, *Argonne National Laboratory*; Lorin Hochstein, *Nimbus Services*

Making Problem Diagnosis Work for Large-Scale, Production Storage Systems27
Michael P. Kasick and Priya Narasimhan, *Carnegie Mellon University*; Kevin Harms, *Argonne National Laboratory*

dsync: Efficient Block-wise Synchronization of Multi-Gigabyte Binary Data45
Thomas Knauth and Christof Fetzer, *Technische Universität Dresden*

HotSnap: A Hot Distributed Snapshot System For Virtual Machine Cluster59
Lei Cui, Bo Li, Yangyang Zhang, and Jianxin Li, *Beihang University*

Supporting Undoability in Systems Operations.75
Ingo Weber and Hiroshi Wada, *NICTA and University of New South Wales*; Alan Fekete, *NICTA and University of Sydney*; Anna Liu and Len Bass, *NICTA and University of New South Wales*

Back to the Future: Fault-tolerant Live Update with Time-traveling State Transfer89
Cristiano Giuffrida, Călin Iorgulescu, Anton Kuijsten, and Andrew S. Tanenbaum, *Vrije Universiteit, Amsterdam*

Live Upgrading Thousands of Servers from an Ancient Red Hat Distribution to 10 Year Newer Debian Based One.105
Marc Merlin, *Google, Inc.*

Managing Smartphone Testbeds with SmartLab115
Georgios Larkou, Constantinos Costa, Panayiotis G. Andreou, Andreas Konstantinidis, and Demetrios Zeinalipour-Yazti, *University of Cyprus*

YinzCam: Experiences with In-Venue Mobile Video and Replays133
Nathan D. Mickulicz, Priya Narasimhan, and Rajeev Gandhi, *YinzCam, Inc. and Carnegie Mellon University*

Friday, November 8

Challenges to Error Diagnosis in Hadoop Ecosystems145
Jim (Zhanwen) Li, *NICTA*; Siyuan He, *Citibank*; Liming Zhu, *NICTA and University of New South Wales*; Xiwei Xu, *NICTA*; Min Fu, *University of New South Wales*; Len Bass and Anna Liu, *NICTA and University of New South Wales*; An Binh Tran, *University of New South Wales*

Installation of an External Lustre Filesystem Using Cray esMS Management and Lustre 1.8.6155
Patrick Webb, *Cray Inc.*

Message from the Program Co-Chairs

Welcome to the LISA '13, the 27th Large Installation System Administration Conference. It is our pleasure as program co-chairs to present this year's program and proceedings.

This conference is the result of the hard work of many people. We thank our authors, shepherds, external reviewers, speakers, tutorial instructors, conference organizers, attendees, and the USENIX staff. We'd particularly like to thank our program committee and the coordinators of the new LISA Labs program for being willing to experiment with new approaches.

This year we have accepted 13 submissions, which adds to USENIX's considerable body of published work. Some of these papers will be applicable for LISA attendees directly, while others are more speculative, and provide ideas for future experimentation. When choosing the program, we specifically chose some content to stimulate discussion and hopefully future paper submissions. Publications in this area—the crossroads of operations practitioners and systems researchers—are difficult to recruit. Framing these complex issues in ways from which others can benefit has always been challenging.

We've prepared a program that we are proud of, but the largest contribution to the success of LISA is the wealth of experience and opinions of our attendees. It is this combination of technical program and deep discussion of deep technical problems that makes LISA unique in our field. We hope that you enjoy the conference.

Narayan Desai, *Argonne National Laboratory*
Kent Skaar, *VMware, Inc.*
LISA '13 Program Co-Chairs

Building Software Environments for Research Computing Clusters

Mark Howison

Brown University

Aaron Shen

Brown University

Andrew Loomis

Brown University

Abstract

Over the past two years, we have built a diverse software environment of over 200 scientific applications for our research computing platform at Brown University. In this report, we share the policies and best practices we have developed to simplify the configuration and installation of this software environment and to improve its usability and performance. In addition, we present a reference implementation of an environment modules system, called PyModules, that incorporates many of these ideas.

Tags

HPC, software installation, configuration management

1 Introduction

Universities are increasingly centralizing their research compute resources from individual science departments to a single, comprehensive service provider. At Brown University, that provider is the Center for Computation and Visualization (CCV), and it is responsible for supporting the computational needs of users from over 50 academic departments and research centers including the life, physical, and social sciences. The move to centralized research computing has created an increasing demand for applications from diverse scientific domains, which have diverse requirements, software installation procedures and dependencies. While individual departments may need to provide only a handful of key applications for their researchers, a service provider like CCV may need to support hundreds of applications.

At last year's LISA conference, Keen et al. [6] described how the High-Performance Computing Center at Michigan State University deployed a centralized research computing platform. Their case study covers many of the important facets of building such a system, including workload characterization, cluster man-

agement and scheduling, network and storage configuration, physical installation, and security. In this report, we look in depth at a particular issue that they touched on only briefly: how to provide a usable software environment to a diverse user base of researchers. We describe the best practices we have used to deploy the software environment on our own cluster, as well as a new system, PyModules, we developed to make this deployment easier. Finally, we speculate on the changes to software management that will occur as more research computing moves from local, university-operated clusters to high-performance computing (HPC) resources that are provisioned in the cloud.

2 Best Practices

As Keen et al. noted, it is common practice to organize the available software on a research compute cluster into modules, with each module representing a specific version of a software package. In fact, this practice dates back nearly 20 years to the Environment Modules tool created by Furlani and Osel [3], which allows administrators to write "modulefiles" that define how a user's environment is modified to access a specific application.

The Environment Modules software makes it possible to install several different versions of the same software package on the same system, allowing users to reliably access a specific version. This is important for stability and for avoiding problems with backward compatibility, especially for software with major changes between versions, such as differences in APIs or the addition or removal of features or default parameters. If only a single version of a software package can be installed at a given time (as is the case for most software package managers for Linux), updating that package to a different version may break users' existing workflows without warning.

While the flexibility introduced by modules is beneficial to both administrators and users, it also creates complexities when modules are used with open-source

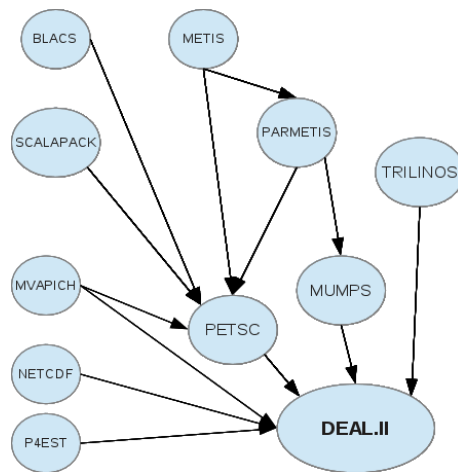


Figure 1: Dependency graph for the deal.II software package.

software.¹ Complexities arise with software configuration and optimization, dependency tracking, and interactions with operating system packages. We address each of these in the subsections below.

2.1 Managing the Configuration and Build Process

Most open-source software installed on CCV’s compute cluster uses one of two configuration and build systems: GNU autoconf² or Kitware’s CMake³ system. Both follow a similar convention that software dependencies are installed in the canonical locations `/usr` or `/usr/local`.

In a research computing software environment with a modules system, this is rarely the case. Instead, dependencies are installed in separate directories in non-canonical locations. Both autoconf and CMake provide mechanisms for specifying the install path of dependencies, but these can lead to very complicated configurations for software with multiple dependencies and sub-dependencies. For example, deal.II [1], a software package installed at CCV and used for analyzing differential equations, has direct or indirect dependencies on roughly ten other software packages (see Figure 1). The dependencies are all installed at different locations in the system. Below is an actual command we used at one point to configure deal.II:

¹Running closed-source software in a modules system is straightforward. Modifying the user’s environment so it contains the path to the executable, and the paths to any license files (if a central licensing server is being used), is usually enough.

²<http://www.gnu.org/software/autoconf/>

³<http://www.cmake.org/>

```

./configure --disable-threads
--with-petsc=/gpfs/runtime/opt/petsc/3.0.0-p12
--with-petsc-arch=linux-gnu-cxx-opt --with-umfpack
--with-trilinos=/gpfs/runtime/opt/trilinos/10.2.2
--with-metis=/gpfs/runtime/opt/metis/4.0.1
--with-blas=goto2 --with-lapack=goto2
--with-p4est=/gpfs/runtime/opt/dealii/7.0.0/p4est
--with-mumps=/gpfs/runtime/opt/mumps/4.9.2
--with-scalapack=/gpfs/runtime/opt/gotoblas2/1.13/lib
--with-blacs=/gpfs/runtime/opt/gotoblas2/1.13/src/BLACS
--enable-mpi CC=mpicc CXX=mpicc
LDFLAGS=-L/gpfs/runtime/opt/gotoblas2/1.13/lib

```

The GNU compilers provide a useful workaround, however, in the form of two environment variables: `CPATH` and `LIBRARY_PATH`. These provide additional directories to search for headers and libraries after the canonical ones, and are also supported by the current versions of other popular compilers, including those from the Intel (2011.11.339) and PGI (12.9) compiler suites. As a result, we specify these variables in any module that contains a library that may serve as a dependency for another library or application. With this setup, complex configuration commands are no longer needed. It is only necessary to have the appropriate dependency modules loaded at compile time.

Setting these environment variables not only makes it easier for administrators to install software packages, but also for users to compile their own custom code. In effect, the variables create an environment that is much closer to the canonical case where all software is installed in `/usr/local` and configuration and building “just works.” However, it retains the added flexibility for upgrading and maintaining multiple versions of software provided by modules.

2.2 Handling Dependencies at Runtime

Software packages that rely on other modules at runtime (for example, dynamically linked libraries) present the same complexity problem at runtime as they do at build time: additional modules must be loaded to satisfy each dependency. This problem can be exacerbated if the software only works with a particular version of a dependency but multiple versions are installed. One possible solution is to include all paths to dependencies in each package’s `LD_LIBRARY_PATH`. However, this leads to modulefiles that quickly grow out of control and present the same readability problems as the deal.II configuration above.

Our preferred solution is to set the `LD_RUN_PATH` variable in the modulefile for any package that provides a library. Then, compiling a dependent package against the library only requires loading the library module at build time. The dependent package will use the library module’s `LD_RUN_PATH` to hard code the absolute path to the library as the default location to search at runtime.

CPU	Highest SSE	# Nodes
Intel Xeon E5540 (Nehalem)	SSE4.2	240
Intel Xeon X5650 (Nehalem)	SSE4.2	34
Intel Xeon X7560 (Nehalem)	SSE4.2	2
Intel Xeon E5-2670 (Sandy Bridge)	AVX	116
AMD Opteron 8382 (Shanghai)	SSE4a	7
AMD Opteron 6282SE (Interlagos)	AVX	2

Table 1: CPU architectures found in CCV’s Oscar compute cluster.

One caveat with using `LD_RUN_PATH` is that moving the library to a different location will break the dependent package. But in a software environment managed by a modules system, the location is typically determined by the package name and version and is rarely moved.

To date, we have successfully used the `LD_RUN_PATH` strategy for all of our library dependencies, even those as complicated as an MPI library (MVAPICH2). The only edge case we have discovered is when a build system passes the `-rpath` flag to the linker. In this case, the `LD_RUN_PATH` value is ignored.

There are two possible solutions. If the hard-coded `-rpath` contains few libraries compared to `LD_RUN_PATH`, the `-rpath` flag can be removed manually and the libraries in it moved to `LD_RUN_PATH`. If `-rpath` contains significantly more libraries, it could be more expedient to add the relevant paths from `LD_RUN_PATH` with additional `-rpath` flags. This is usually as easy as editing a single configuration file, provided the software is using a standard build system. If the software is making use of `-rpath` in a non-standard build system and it is impractical or too difficult to change, then use of `LD_LIBRARY_PATH` should be considered.

2.3 Performance Optimization

On homogeneous clusters, where every node shares the same CPU architecture, software can be compiled using a host-specific optimization flag, such as `-march=native` for the GNU compilers or `-fast` for the Intel compilers. Additionally, many important math libraries have optimized, vendor-provided implementations, such as the AMD Core Math Library⁴ or the Intel Math Kernel Library⁵.

In other cases, however, optimization is not as straightforward. Because hardware procurement can happen in small cycles – especially under a model in which investigators write equipment funding into grants to contribute

to a community cluster – a university research cluster can evolve into a heterogeneous mixture of hardware. At Brown, our research computing platform, Oscar, includes nodes with five similar but distinct CPU architectures, summarized in Table 1. This is in contrast to large, homogeneous systems installed at larger centers like those run by the DOE and NSF.

Conceivably, fully optimizing the software environment to take advantage of the different architectures in a cluster requires localized installations that are specific to each architecture. We have experimented with this at the following three levels of granularity:

- At the coarsest level, using the processor vendor: either Intel or AMD. This allows software to be compiled against the Core Math Library for AMD processors, and the Math Kernel Library for Intel processors.
- At a finer level, using the highest vector instruction set supported by the node. This allows software to take advantage of such instructions if they are available on a node, or otherwise to fall back to a version that works on all nodes.
- At the finest level, using the precise model of processor on the node. This can be used when installing software packages that use autotuning to optimize themselves at build time for a specific processor model.

In practice, though, we have found all of these lacking. Because the Intel Math Kernel Library performs well on all of our AMD and Intel nodes, and automatically selects processor optimizations at runtime, we have abandoned using processor-specific or autotuned alternatives like GotoBLAS [4] and ATLAS [2]. Therefore, neither the first or third levels of localization are necessary.

The second level of localization for the vector instruction set, has not been useful because the most widely used instructions are already available in the older set (SSE3) that is common to all the processors in our cluster.⁶ In one example, we expected several bioinformatics packages that mainly perform string comparisons to benefit from compiling with support for the packed string comparison instructions added in a newer instruction set. What we found instead was that these programs implicitly used the newer instructions through calls to standard C string functions, which are implemented in `glibc` with a mechanism to auto-detect the available instruction set at runtime. Therefore, compiling separate versions for different instruction sets is not necessary.

All of the approaches to performance optimization listed above create a more complicated software environ-

⁴<http://developer.amd.com/tools/cpu-development/amd-core-math-library-acml/>

⁵<http://software.intel.com/en-us/intel-mkl>

⁶For a more detailed comparison of benchmarks across different instruction sets, see <https://bitbucket.org/mhowison/pymodules/src/master/npb-test>.

ment, by requiring multiple versions of a module to support different hardware profiles. Because we have not seen significant gains from them in most cases, we use these optimizations sparingly.

Overall, our optimization strategy has devolved to simply using generic flags like `-O3` and `-msse3` during compilation for most modules.

2.4 Operating System Packages

Much of the software we build depends on libraries provided by operating system packages (from CentOS, in our case). Because the compute nodes in our cluster are diskless, their operating system is installed to a ramdisk. To reduce the footprint of this ramdisk, we install only a minimal set of packages on the compute nodes. We install a fuller set of packages, including development versions of packages, on the login nodes where we compile software. To satisfy runtime dependencies, though, we have to copy many of the libraries provided by the OS packages into a special module, `centos-libs`, that is loaded by default by all users. For most packages, we can use a simple script that parses all of the shared library names from the package's file list and copies them into `centos-libs`. Inevitably, some packages have further complications, for instance because they require additional data in a `share` directory, and we handle these on a case-by-case basis by manually copying additional files into `centos-libs`.

We have only performed one major OS upgrade (from CentOS 5.4 to 6.3), and in this case we chose to rebuild our entire software environment against the newer OS version. While this required substantial effort, it also provided us with two opportunities: (1) to verify how well documented our build processes were, and correct modules that were poorly documented; and (2) to weed out older modules that were no longer in use by our users.

3 PyModules

PyModules is an alternative implementation of the Environment Modules system [3]. We have given PyModules essentially the same syntax and user interface as Environment Modules; however, the backend is written in Python and contains the following improvements:

Simple, INI-style configuration files.

Having created many modulefiles in the Tcl language for the original Environment Modules system, we found they were unnecessarily redundant, since a new file is created per version of an existing package. We designed PyModules to instead use a single INI-style configuration file per package. Multiple versions are defined in

that same file and they can inherit default values, which has simplified our management of these files.

Below is an excerpt from our configuration file for Python, which defines three versions:

```
[DEFAULT]
brief = The Python Programming Language
url = http://www.python.org/
category = languages

prepend PATH = %(rootdir)s/bin
prepend LIBRARY_PATH = %(rootdir)s/lib
prepend LD_LIBRARY_PATH = %(rootdir)s/lib

[2.7.3]
default=true
[3.2.3]
[3.3.0]
```

Our approach gives up some flexibility, since the INI modulefiles cannot execute arbitrary Tcl commands. However, it has the added benefit that we can validate each modulefile before it is available to users. In Environment Modules, it is possible to create modulefiles that generate a Tcl parsing error when loaded.

Improved inventory commands.

Users often need to perform software inventory commands, such as looking up what versions of a specific software package are installed. In PyModules, we cache the parsed modulefiles in an SQLite database, which speeds up both the `module avail` command and the `autocomplete` feature of the `module` command. We also create a `fulltext` index to support wildcard searches on package names and versions. For example, the command `module avail mpi` will show all available modules that start with the token `mpi` in the package name, and the command `module avail mpi/1` will additionally filter only the versions with a 1 in them.

The database is manually updated whenever a new INI configuration file is created or modified, using a new command called `moduledb`. This allows an administrator to review and validate a modulefile before committing it to the live system.

Module categories.

PyModules provides a special field `category` in the INI configuration that can be used to categorize the available modules into bioinformatics packages, physics packages, chemistry packages, etc. The command `module avail` lists all packages broken down by category, and the new command `module avail :category` lists only the packages in the specified category. Modules can belong to multiple categories.

4 Software in the Cloud

Looking forward, cloud-based HPC resources are promising alternatives to university-operated clusters, especially at the scale of small or departmental clusters [5]. Such clusters will require support for applications running on virtualized hardware. This could alleviate many of the problems we have described here, because virtualization provides a new layer of flexibility in customizing the operating system. Instead of providing a single software environment that must span researchers from many disciplines, software environments can be customized by department or research field, and deployed through OS images. Even traditional HPC clusters may in the future provide more support for users to run their own custom OS images.

Shifting the unit of software organization from a software module to an OS image has many implications:

- Administrators will manage a catalog of images for the most common applications. The same mechanisms for versioning and inventory will be needed, but at the OS-image level instead of the module level.
- Versioning of individual software packages may no longer be necessary: instead, an entire OS image can be versioned. This also reduces the complexity of dependencies, since they can be packaged together in the image.
- Since it is no longer necessary to version the software, it could be installed by more canonical means, such as directly to `/usr/local` or using the native OS package manager.
- Tools for automating configuration and builds will become essential, as common packages are re-installed in many OS images.

An important question for building software environments in the cloud is: will the overhead from virtualization degrade application performance? Our experience with the fine-grained optimizations we described in Section 2.3 seems to suggest not: modern software is increasingly using and benefitting from the ability to leverage hardware optimizations, such as vectorization, at runtime. This is corroborated by a study of HPC benchmarks showing that aggressive tuning of the hypervisor can reduce its overhead to only 1% to 5% [7].

Finally, a shift to OS images has important benefits for software distribution. Especially in bioinformatics, there are large-scale challenges to scientific reproducibility caused by incomplete software tools that are difficult to install or use by novice end users. We frequently receive support requests at CCV to install new bioinformatics tools that have complicated dependencies or incomplete build systems. Moving to a paradigm where exper-

imental tools are available via OS images is perhaps the best solution to this problem [8].

5 Conclusion

This report documents the best practices we have arrived at for installing a large collection of scientific applications on our research computing cluster at Brown University. In this context, we have also introduced a new implementation, PyModules, of the Environment Modules system that has helped improve the usability of our software environment. Finally, we have identified possible changes to software management practices resulting from the OS-level virtualization available in the cloud.

Acknowledgments

We thank Paul Krizak for his thorough feedback, which improved the clarity and presentation of the paper.

Availability

PyModules is freely available under a non-commercial license from:

<https://bitbucket.org/mhowison/pymodules>

References

- [1] BANGERTH, W., HARTMANN, R., AND KANSCHAT, G. deal.II - A general-purpose object-oriented finite element library. *ACM Trans. Math. Softw.* 33, 4 (2007).
- [2] CLINT WHALEY, R., PETITET, A., AND DONGARRA, J. J. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing* 27, 1-2 (2001), 3-35.
- [3] FURLANI, J. L., AND OSEL, P. W. Abstract Yourself with Modules. In *Proceedings of the 10th USENIX System Administration Conference (LISA '96)* (Chicago, IL, USA, Sept. 1996).
- [4] GOTO, K., AND GEIJN, R. A. V. D. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.* 34, 3 (2008), 12:1-12:25.
- [5] HILL, Z., AND HUMPHREY, M. A quantitative analysis of high performance computing with Amazon's EC2 infrastructure: The death of the local cluster? In *Proceedings of the 10th IEEE/ACM International Conference on Grid Computing* (Banff, AB, Canada, Oct. 2009), pp. 26-33.
- [6] KEEN, A. R., PUNCH, W. F., AND MASON, G. Lessons Learned When Building a Greenfield High Performance Computing Ecosystem. In *Proceedings of the 26th USENIX Large Installation System Administration Conference (LISA '12)* (San Diego, CA, USA, Dec. 2012).
- [7] KUDRYAVTSEV, A., KOSHELEV, V., PAVLOVIC, B., AND AVETISYAN, A. Virtualizing HPC applications using modern hypervisors. In *Proceedings of the 2012 Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit (Federated-Clouds '12)* (New York, NY, USA, 2012), ACM, pp. 7-12.
- [8] NOCQ, J., CELTON, M., GENDRON, P., LEMIEUX, S., AND WILHELM, B. T. Harnessing virtual machines to simplify next-generation DNA sequencing analysis. *Bioinformatics* 29, 17 (2013), 2075-2083.

Matt Provost
Weta Digital
Wellington, New Zealand
mprovost@wetafx.co.nz

Tags: Nagios, monitoring, alerting

Fixing On-call, or How to Sleep Through the Night

ABSTRACT

Monitoring systems are some of the most critical pieces of infrastructure for a systems administration team. They can also be a major cause of sleepless nights and lost weekends for the on-call sysadmin. This paper looks at a mature Nagios system that has been in continuous use for seven years with the same team of sysadmins. By 2012 it had grown into something that was causing significant disruption for the team and there was a major push to reform it into something more reasonable. We look at how a reduction in after hour alerts was achieved, together with an increase in overall reliability, and what lessons were learned from this effort.

BACKGROUND

Weta Digital is Sir Peter Jackson's visual effects facility in Wellington, New Zealand. Because of the workflow at Weta, users are at work during the day but our 49,000 core renderwall is busiest overnight when none of the sysadmins are around. So often the highest load at the facility is overnight which is when problems are most likely to occur.

The monitoring system at Weta Digital uses the open source Nagios program and was introduced into the environment in January 2006 following the release of "King Kong". Nagios was replaced with the forked project Icinga in September 2010 but the core of the system has remained the same.

The Systems team at Weta is on a weekly on-call rotation, with 6-8 admins in the schedule. However some admins had configured specific systems to page them directly, and some alerts were configured in Nagios to always go to a specific person, possibly in addition to the on-call pager. So some sysadmins were essentially permanently on-call.

By 2010 the system had become unsustainable. The weekly on-call rotation was dreaded by the sysadmins. Being woken up every night while on-call was not unusual, in fact getting a good night's sleep was remarkable. Being woken up several times during the night was so common that we had to institute a new policy allowing the on-call sysadmin to sleep in and come to work late when that happened. However many (if not most) of the alerts overnight were non-critical and in many cases the admin would get the alert and then go back to sleep and wait for it to recover on its own, or wait to fix it in the morning.

Some of the admins were starting to compensate for bad on-call weeks by leaving as early as lunchtime on the following Monday once they were off call. So the average number of hours across the two weeks of on-call and the next week still averaged out to two normal weeks, but it was impacting on their project work by essentially missing a day of work and often left the team shorthanded when they were out recovering from on-call.

Sysadmins at Weta are independent contractors on an hourly rate. Each after hours alert therefore creates a billable hour. It could be construed that there is a financial incentive for creating new alerts, whether people think about it that way or not. The team itself is responsible for maintaining the Nagios system and adding alerts, so asking them to try and reduce the number of alerts, thereby reducing their income, was always going to be an unpopular move.

NAGIOS

One of the differences with the Weta system is that it uses a set of configuration files that are processed by a Perl script to generate a Nagios configuration. This is used to manage the most common configuration changes via a single file, `hosts.txt`, which manages both host and service checks. The `hosts.txt` file is similar in format to the original Netsaint configuration. (Netsaint was the original name of the Nagios software in 1999 but was changed in 2005 for trademark reasons.) The original Netsaint configuration files consisted of lines with semicolon delimited fields:

Format:

```
host[<host_name>]=<host_alias>;<address>;<parent_hosts>;<host_check_command>;<max_attempts>;<notification_interval>;<notification_period>;<notify_recovery>;<notify_down>;<notify_unreachable>;<event_handler>
```

Example:

```
host[es-gra]=ES-GRA Server;192.168.0.1;;check-host-alive;3;120;24x7;1;1;1;
```

The Weta configuration files look similar, but with colons separating the fields:

Format:

```
<host_name>:<host_type>:<check_name>:<check_command>:<parent_host>
```

Examples:

```
imap::check_long_running_procs,nosms:check_long_running_procs:vrrpe120
adsk-license2:Linux:check_autodesk_is_running:passive:vrrpe120
ldapauth0:Linux:ldap:check_ldap!ou=People,dc=wetafx,dc=co,dc=nz:vrrpe120
```

Because the config is reduced to one line per command it is trivial to add large new sections of checks with simple for loops. This contributed in part to the proliferation of checks in the configuration. In some cases there are specific scripts that are used to generate hundreds of checks.

Most of the specifics for each host and service check are filled in via the Perl script using template files. The config is stored in RCS which has allowed us to review and analyse the history of the configuration since 2006.

THE PROBLEM

Over the years the amount of hosts being monitored by Nagios has grown from an initial 297 to 843 today, with a peak of 915 in 2011. The number of service checks has grown at a much higher rate, from 365 initially to 3235 today (April 2013), with a peak of 3426 in December 2012.

How did the number of checks grow so much over time? For the most part it was just the natural accumulation of new checks being added, and then retroactively being added to other hosts. So adding a new check to a Linux host that then gets applied to all of the others can easily add hundreds of checks. There were a few incidents like this where there were step changes and many checks were added, but there was also just the slow growth of adding servers. Moving much of our server infrastructure to VMware has certainly increased the number of individual hosts in use. Of course over time there are also an increasing number of legacy systems to maintain.

Starting in October 2010, in response to a growing number of complaints from the team about on-call, I began a project to reduce the number of checks that would alert the Systems team, by increasing the number of alerts that were configured not to send pages. However nothing was done to reduce the number of checks themselves. The number of email problem notifications was still averaging over 50 per day. In fact there were so many alerts that were being ignored that some started to be labelled with the suffix ‘_fix_asap’ to distinguish them from all of the noise. That was a critical sign that the system was broken. People were configuring Nagios to alert for ‘their’ systems to make sure that they were being looked after. But this meant that some systems were being aggressively alerted, more than necessary.

Ignored alerts are false alarms and are Nagios itself notifying us that it was really the system that was broken. And based on that, our most unreliable system was Nagios itself. So it needed a major overhaul. The amount of alerts going to the on-call sysadmin and the number of emailed alerts going to the whole team were still overwhelming, so in January 2012 I started another project for the team to clean up the Nagios configuration.

CLEANING UP

One of the features of the original Perl template system is called ‘nosms’. This allows the admin to specify that (originally) a particular host or (later) a service check should not send an SMS alert to the on-call sysadmin. This was designed to keep non-mission-critical systems from alerting anyone out of hours.

However adoption of the nosms option was very low - it started with 2 hosts in January 2006 and was only up to 30 by January 2011. Under pressure to improve the on-call situation it had increased to 202 a year later in 2012, but with a concerted effort it is now at 807 checks. Along with other changes to the system to stop overnight alerts, the number of checks that can wake someone up is down to 1280 from a high of 1763. That number is still too high and work is ongoing to reduce it further.

Individual members of the team had also configured Nagios to page them specifically when ‘their’ systems were having issues, even when they weren’t on-call. These have all been removed, so now when you are not on-call you won’t be bothered. Even though people were doing this to be helpful and look after the systems for which they are responsible, it just creates burnout for those individuals. The basic rule now is, when you’re not on-call, you don’t get paged.

I also instituted a new policy, that the on-call sysadmin has to call the vendor before waking up someone else on the team. Due to the small size of the team and the large number of different systems that we have to support there is some inevitable siloing within the team where certain people look after certain systems. The tendency is to call those people first when ‘their’ system is in trouble, but that just penalises an admin who has to look after the more unreliable systems, even if there is nothing that they can do to improve the situation. Some hardware is just less reliable than others, and unfortunately you don’t often discover that before you buy it. This was creating an environment where people were afraid to express interest in and thereby become responsible for some new equipment in case it turned out to be unreliable. Weta pays a significant amount of money each year for support contracts so that we can call tech support 24/7. So the new rule places more of a burden on the vendors and less on other team members.

One change that we made as a team was to move to a consensus based approach to adding new alerts. So new alerts have to be discussed and if even one person doesn’t think that it’s important enough to wake up for then it isn’t added to the configuration. Whoever wants to add the alert has to convince everyone else on the team that it is worth waking up for. This stops people from adding extra alerts for specific systems without consulting with the rest of the team. It’s important to do this in an open, non-confrontational way, preferably in person and not over email where discussions can get heated. It has to be a positive environment where senior members of the team can’t intimidate more junior members to go along with their ideas. Management also has to provide input and get feedback from the stakeholders of a particular system to make sure that business requirements are still being met.

Nagios shouldn’t be used as a means of avoiding work in the future. Priority needs to be placed on not bothering the on-call sysadmin with problems that can be deferred until the next working day. In some cases systems could be configured to run overnight but it would create a mess that someone would have to clean up in the morning. There is a natural tendency for the person who will most likely have to clean things up to make an alert and wake someone up to deal with the problem before it gets severe and creates a situation where there would be more effort involved to clean it up after the fact. In that case there is only a 1/8 chance that that person responsible for that system would be the one on-call, but they would always have to clean things up in the morning. So the natural reaction was to create an alert. Instead we should be placing a higher importance on respecting people’s time (and sleep) when they are on-call so doing more work during office hours is preferable to alerting someone out of hours.

In our Nagios configuration some systems were identified by multiple DNS CNAMEs. This would happen because we often separate the system's hardware name from a more functional name, but then both would be monitored. This would result in duplicate alerts, although it usually wasn't until you logged in and checked that you would realise that they were for the same system. It also created confusion during more critical events like a network outage where Nagios wasn't correctly reporting the actual number of hosts that were affected. So we went through a cleanup and rationalised all of the hostnames down to a unique set.

THE LIST

By January 2012 tensions around on-call reached a breaking point. I decided that my earlier efforts to gradually clean up the system were happening too slowly - for every alert that was being nosms'd or deleted, two more new alerts were created. I began by consulting with the CIO, who was shocked at the amount of monitoring email alerts in her mailbox, and coming up with a list of systems that were deemed critical to the facility and should be monitored 24/7. This list was quite small in comparison - less than 50 hosts as compared to the over 1600 in Nagios at the time.

This led to a time period that I call the 'Constitutional Phase' because of how the list was pored over for inconsistencies by the team as if it was a legal document. Faced with an extremely shortened list of hosts, the team started to question each decision as to why certain hosts were on or off the list. For example, "If host A is on the list, host B is just as important so it should be on there as well." Or "If host C isn't on the list then D and E shouldn't be on there either."

Rather than trying to articulate and justify each decision individually I took a new approach which was to try and come up with a flow chart or decision tree that would have a consistent set of questions to determine whether each host in the Nagios config would continue to alert or not, eventually resulting in the much shortened list. One example of such a list of questions was:

Is it a critical production system?

- No dev/test/stage

- No backup/offline/nearline

Is it the renderwall?

- Will it affect more than 50% of the renderwall?

Is it a critical piece of the Production pipeline?

- Dailies/Filemaker/email

Will it affect clients?

- Aspera/SFTP/external website/external email

Will it affect all users of a piece of core render software?

- license servers

However in an attempt to try and limit the list to things that would be considered major outages, I was falling into the trap of setting thresholds (will it affect more than 50% of the render servers). So I was never very happy with this approach and it was never implemented.

The next attempt was to focus on who was affected:

Will it affect:

- Dailies?

- Core render software for the renderwall

- Users?

- Core software for users/email/Filemaker

- Clients?

- Aspera/SFTP/email/website

But this would lead to keeping almost everything on the list because almost every system affects renders, users or clients. So again it was never implemented.

The final approach that I took was to simplify the list of questions to the most basic ones:

Is it:

- A production system (no dev/test/stage) and
- A primary system (no backup/offline/nearline)
- or a single point of failure for another critical system

This does pretty much lead to an unchanged list with the exception of removing things like dev systems (yes, before this dev and test systems would normally alert). However there was a fourth question that I added which changed the whole focus of the project:

Is it configured to automatically reboot/restart/repair itself?

I realised that focusing on Nagios itself wasn't the goal. Only a certain amount of non-critical systems could be either removed or set not to alert after hours, but we weren't really fixing anything but Nagios itself. The number of alerts coming in wasn't really changing, it was just that fewer of them were going to the on-call person's phone. The team's workflow consisted of finding a problem and then adding a Nagios alert for it, and never actually going back and resolving the underlying issue. Once it was in Nagios it was considered 'fixed' even though it kept alerting and was being resolved by hand. We needed to change our focus from just monitoring systems to actually fixing them.

Once I had a new list, which was actually now just a set of questions, I ran it past the CIO to make sure that I had her support before continuing. It was important to get her approval to make sure that it would meet the business' needs, and she agreed with the direction that the project was taking, even allowing some decisions which the team was convinced would lead to reduced service levels. However, she highlighted the need for striking the right balance through trial and error.

FIXING

Nagios was being used as a job scheduler to get sysadmins to perform some task. Alerting had become the culture of the team and was viewed as the solution to every problem. Where possible all of those tasks should be automated instead.

As an example, we had Nagios alerts set up to tell us when there were security updates available for servers in our DMZ network. Since they were all running the same version of Debian you would get a flood of identical alerts at the same time as a new update became available. This was configured as a nosms alert, but someone on the team would see the emails and manually install the updates. This has now been replaced with apticron which downloads and installs the updates automatically. The Nagios check remains to alert us when this process fails and human intervention is still required.

Before 2009 there was nothing configured to automatically restart failed programs. Starting in 2009, after a string of alerts from an unreliable daemon, we began using DJB's daemontools to automatically restart critical services if they crashed. It's now a rule that we don't monitor anything unless it's configured to automatically restart, where possible. This may seem counterintuitive - we only monitor services that have already been configured to restart. But this places the emphasis on fixing the problem and not just putting in a Nagios check and letting someone deal with it at a later date.

To resolve unresponsive (hung) Linux servers, we first started configuring the kernel to reboot after a panic. By default the system will wait with the panic message, but we're just interested in getting it back up and running. Later in 2012 the kernel watchdog was also enabled. This will automatically reboot a server if the kernel stops responding. In the case where a server is hung there is typically nothing for the on-call person to do but use the out-of-band management to hard reboot the system. Now this happens automatically before a Nagios alert is sent out. For our

servers in VMware there is a monitoring setting available that will automatically restart servers if they are unresponsive and not doing IO.

Weta has many Netapp filers for NFS file serving and sometimes they panic due to software bugs. They come in redundant pairs, but the on-call admin would be notified that one node was down and would log in and reboot it and fail back the cluster. There is a configuration option for automatic cluster giveback which we have enabled on all of our filers now, so unless it's a hardware fault the cluster can automatically recover itself and not alert.

We were also managing disk space manually on the filers before this project. So an alert would be sent notifying us that some filesystem was getting close to capacity, and the on-call sysadmin would log in and extend the filesystem (if possible) or delete some files. We have enabled the filesystem autosize feature where the Netapp will automatically add space to filesystems as they get close to filling up, up to a specified limit. Now the on-call person only gets paged when it hits that limit after expanding several times automatically. This has eliminated most of the overnight disk space management.

Logfiles filling disks was another thing that would wake people up. Now that we're focusing on fixing the alerts, automatic log rotation is set up to stop disks from filling up overnight instead of manually deleting old log files in the middle of the night. Any alert coming in from a log that hasn't been set to rotate should be resolved by adding it to the logrotate configuration.

If you buy a redundant system you should be able to trust it. So we've stopped alerting on failed disks in our filers, and even failed heads since they're part of a redundant clustered pair. We always configure the filers with plenty of hot spare disks. The same rule goes for systems with redundant power supplies or anything else that is designed to keep functioning when one component fails. You are taking the risk that possibly a second component will fail overnight but in practise that is extremely unlikely. One of the reasons for buying redundant hardware is to keep your admin team from being bothered by every failure if the system can keep running.

In the case of load balanced services, instead of alerting on each server behind the load balancer, the Nagios checks have to be rewritten to do an end-to-end check of the service. It doesn't matter if one server fails since the load balancer will hide that from clients. As long as one server is up and running, no alerts need to be sent. If the load is too great for the remaining servers then the service check should timeout or fail and alert the on-call sysadmin as usual.

One phenomenon that I was seeing I call 'Twikiscript' after the wiki that we use for team documentation. There is a page 'NagiosAlertsAndErrorsAndWhatToDo' which contains sections for about 150 different Nagios alerts. Many of them contain instructions that are in the form of "log in, look for this and do this to resolve it". Some of the sections include branches and conditionals, and in essence are scripts written for humans to execute. Most of them could be rewritten as scripts for computers to run with a little work.

For example, one alert that used to reliably wake people up several times a week was sent when our MySQL replication slaves would start to lag behind the master. This could be for a variety of reasons, typically high load or bad queries. The fix, following the instructions in the wiki, was to log in and update the parameter in MySQL that would change the InnoDB engine from the standard behaviour of flushing data to disk after each transaction to only doing it once per second. We call this 'turbonating' the server. This creates the risk that you can lose up to a second's worth of data but also increases performance and typically allows the slaves to catch up. Because they were slave copies, the DBA team was ok with us taking this risk. Often you would get woken up, change this setting, go back to sleep, wait for the recovery alert and then change it back. Now we've created a cron job that watches the lag every 5 minutes and if it's beyond a threshold in seconds it changes the MySQL parameter to the faster setting. If the lag is below the threshold it changes it back to the safer setting. It also emails out so that the DBA team will know that it has happened and investigate possible causes. This is called 'autoturbonation' and since it has gone in has completely eliminated the overnight alerts.

We've also started to use the swatch program to help fix problems automatically. It's a Perl program that tails a log file and searches each line for regular expressions and then can execute a script when it finds a match. It can be used

for notification, so we configure it to email the team when certain errors appear. We also use it to fix things automatically, so when a recurring error shows up that has a known fix, we can script that fix, hook it into the swatch configuration and stop getting alerts.

Alerts should be actionable, and in a reasonable timeframe. We monitor SSL certificate expiry in Nagios, with a 30 day warning to give us time to order a new cert. But typically what happens is that the alert goes off daily for about 28 days and then someone will get around to it.

Another example of non-actionable alerts are the checks for batteries on RAID controllers, which expire after a certain amount of time. This is usually known in advance (they have a service life of a year in most cases) but we use Nagios to alert us when the controller shows an error condition due to the battery expiring. But then an order has to be placed with the manufacturer so it can take weeks to replace a battery. This isn't a good use of an alerting system. Ideally they would be replaced on a fixed schedule which would be maintained in a separate calendaring system, along with reminders for renewing SSL certificates with a fallback alert that goes off shortly before they expire.

One way that we keep everyone focused on fixing Nagios errors is to meet weekly on Mondays to go over all of the Nagios emails from the previous week. That week's on-call sysadmin leads the meeting and goes through each alert and what the response from the team was. In many cases there was no response and the system recovered on its own, so we discuss whether we can change the thresholds in Nagios to avoid being notified in the future. Or if it's a recurring issue that we see happening week after week someone can be assigned the task of fixing the underlying issue. With the Nagios configuration in RCS we can also review any commits made during the past week and make sure that the entire team is ok with those changes. If there are any objections the change can be reverted. This meeting also serves as a critical safety check to ensure that we aren't changing the system too much and drifting over the line where we're not picking up events that are critical to the facility by providing an opportunity for anyone to raise problems with the monitoring system.

THE RESULT

As a result of all of these changes, the number of out of hours alerts for the on-call sysadmin dropped significantly. During the production of "Avatar" and "Tintin", people were sometimes working over 70 hour weeks and doing over 60 hours was common when on-call. During the production of "The Hobbit" last year, no one on the team worked over 60 hours due to being on-call. Now sleeping through the night when on-call is the norm, and getting woken up an unusual situation. And as time goes on it is less and less tolerated by the team so there is an acceleration to the process of eliminating recurring alerts. This has dramatically improved the work/life balance of the sysadmin team by reducing out of hours alerts.

The level of the service provided to the facility didn't decrease, in fact the system is the most stable it has ever been. I attribute at least part of this to the work that was done in trying to reduce alerts by fixing long standing problems and configuring them to fix themselves whenever possible instead of waiting for systems to break and reacting to alerts. Computers can respond much quicker than humans and can often correct problems faster than users notice the problem in the first place.

As the CIO has acknowledged, "Ultimately, major cultural and working-practice changes were involved; a shift of emphasis to proactive and preventative maintenance and monitoring, rather than reactive problem solving; greater teamwork and shared ownership of systems rather than solo responsibility for them; and the adoption of a dev-ops approach to system administration with a greater emphasis on automation, standardisation and continuous improvement. Although it is taking some time and tenacity to get there, the results have been overwhelmingly positive and beneficial."

THE FUTURE

One of the improvements that could be made to simplify the Nagios configuration would be to split it into two systems - one for sending pages to the on-call sysadmin and another for the non-critical alerts. This has been accomplished with the nosms feature of our current infrastructure but it adds unnecessary complexity.

We are looking at integrating Nagios into our ticketing system, RT. This would involve opening a ticket for each Nagios alert and closing the issue when Nagios sends the OK. Any work done to resolve the issue can be recorded against the ticket. This would let us track which issues are being worked on by the team and which are resolving themselves (false alarms) so that we can work on eliminating them.

We're starting to use pynag, a Python module for parsing Nagios configurations, to monitor Nagios itself. For example, we can use it to resolve the hostnames in the configuration and alert us when there are duplicate CNAMEs. Or it can verify that no hostnames ending in '-dev' are set to alert. By setting up rules in code it stops the configuration drifting from the policies that have been set by the team.

The final frontier is integrating the Nagios configuration into our Puppet rules when we configure servers. Then we can generate a configuration end to end. So for example we could specify a service in Puppet and configure it to run under daemontools and then automatically add that service to Nagios. If it isn't configured in Puppet then it won't be added to monitoring. This way we can enforce that it is configured to automatically restart.

Monitoring systems are good at maintaining a list of things to check. In the early days implementing one can be a big improvement over being surprised by finding that critical systems are down or in trouble. But in a complex system you end up enumerating every possible component in the monitoring system which is an almost endless list as you chase the long tail of increasingly unlikely conditions to monitor. It's a reactive process where checks are only added as things break. In the long run the only way out is to approach monitoring from a more systemic point of view, looking at the infrastructure as a whole as opposed to a group of component parts.

For example, in Weta's case, we still have never monitored renders, which are the core function of our compute infrastructure. So every render could encounter some condition that causes it to fail overnight and noone would have any idea until the next morning, if it's something that we haven't accounted for in our Nagios checks. A better check would be to have an automated program reading the logs coming off the renderwall and alerting when something out of the ordinary was going on. At that point Nagios could be used by the sysadmin team to triage and try to narrow down where the error was coming from. But in the long run it should be less and less a part of everyday life for a system administrator.

LESSONS LEARNED

Anytime you go beyond the capabilities of your monitoring system I think you have to take a good look at what you're doing and question whether it is truly necessary. We wrote a front end to the Nagios configuration which made it easier to add new checks, and then proceeded to add so many that we were overwhelmed. And we built the nosms feature instead of realising that Nagios is for alerting and that moving non critical checks into another system altogether would be better. In fact we had to make policies for people to be able to sleep in instead of questioning why they weren't getting any sleep in the first place and whether that sacrifice was worth it. Building a custom monitoring system is probably not the best use of a team's time when working on fixing problems can lead to better results.

Unfortunately if there is money involved then no matter how much individuals might complain about being on-call there is always a financial incentive to work more out of hours. Different people on the team have different tolerances for this based on their personal and family situations which can create tension within the team. Trying to fix the problem of on-call can be perceived as an attack on their paycheck and there is no financial incentive for the team to clean up the broken system so it takes some strong leadership from management initially to make it happen. My experience is that once the changes have been made people start to appreciate that it makes a positive improvement in their personal lives and the focus on money fades.

Monitoring systems can grow over time to cover every possible part of the infrastructure, out of a drive from the team to keep everything running all the time, or as reactions to individual system failures that weren't caught by an alert and were reported by users. However this can lead to a 'tree falling in the forest with no one around' situation where sysadmins get alerted over the weekend about some system that wouldn't be noticed until Monday morning. If it's not mission critical it is possible to step down from that level of service and wait for a user to complain instead of proactively fixing every problem. This takes good backing from management so that team members don't feel individually responsible for these decisions. But in our experience it has worked out well for both users and sysadmins.

Because monitoring is a way of assigning tasks to the team, it is a backdoor way for members of the team to work around whatever management systems you may have in place for assigning daily jobs. Moving to an open, consensus-based approach for maintaining the monitoring configuration makes sure that everything is done the way management wants to set the team's priorities.

Some people would rather fix things before they turn into bigger problems and don't want to defer the work until later. If you want to preserve their sanity and work/life balance there has to be a focus on getting things done during work hours and building systems that can run overnight or for a weekend without any human intervention. Each call that someone gets should be a unique problem brought about by some unexpected circumstance or failure. Any routine alert should be fixed with urgency so that it doesn't further bother people outside of work hours.

SUMMARY

- Get management/stakeholder buy in
- No personal alerts (when you're not on-call you don't get paged)
- Don't optimise alert configuration (no custom scripts to make alerting easier, a little friction is good)
- Keep the config(s) in version control for easy tracking and review
- No priority labels - every alert should be critical (no `_fix_asap...`)
- Split monitoring system into critical and noncritical (at least by email address)
- Call vendors before bothering teammates
- Consensus based process for adding new alerts (can be done retrospectively in the weekly meeting)
- Prioritise in hours work over after hours (even if it will take longer)
- No CNAMEs/duplicate hosts
- Change process checks to process monitors with automatic restarting (daemontools, monit, runit...)
- Automate server updates (apticron, up2date...)
- Reboot after panics (`echo "30" > /proc/sys/kernel/panic`)
- Kernel watchdog (man 8 watchdog, VMware monitoring)
- Netapp filer cluster automatic giveback (`options cf.giveback.auto.enable on`)
- Netapp automatic volume expansion (`vol autosize /vol/myvolume/ -m 30g -i 5g on`)
- Automatic log rotation and compression (logrotate)
- No individual alerts for load balanced servers/services (end to end service checks instead)
- Turn Twikiscript into automated processes
- MySQL autoturbonating (`set global innodb_flush_log_at_trx_commit = 2`)
- swatch for finding problems and scripting fixes
- Ticket system integration
- pynag to monitor the Nagios configuration for policy violations

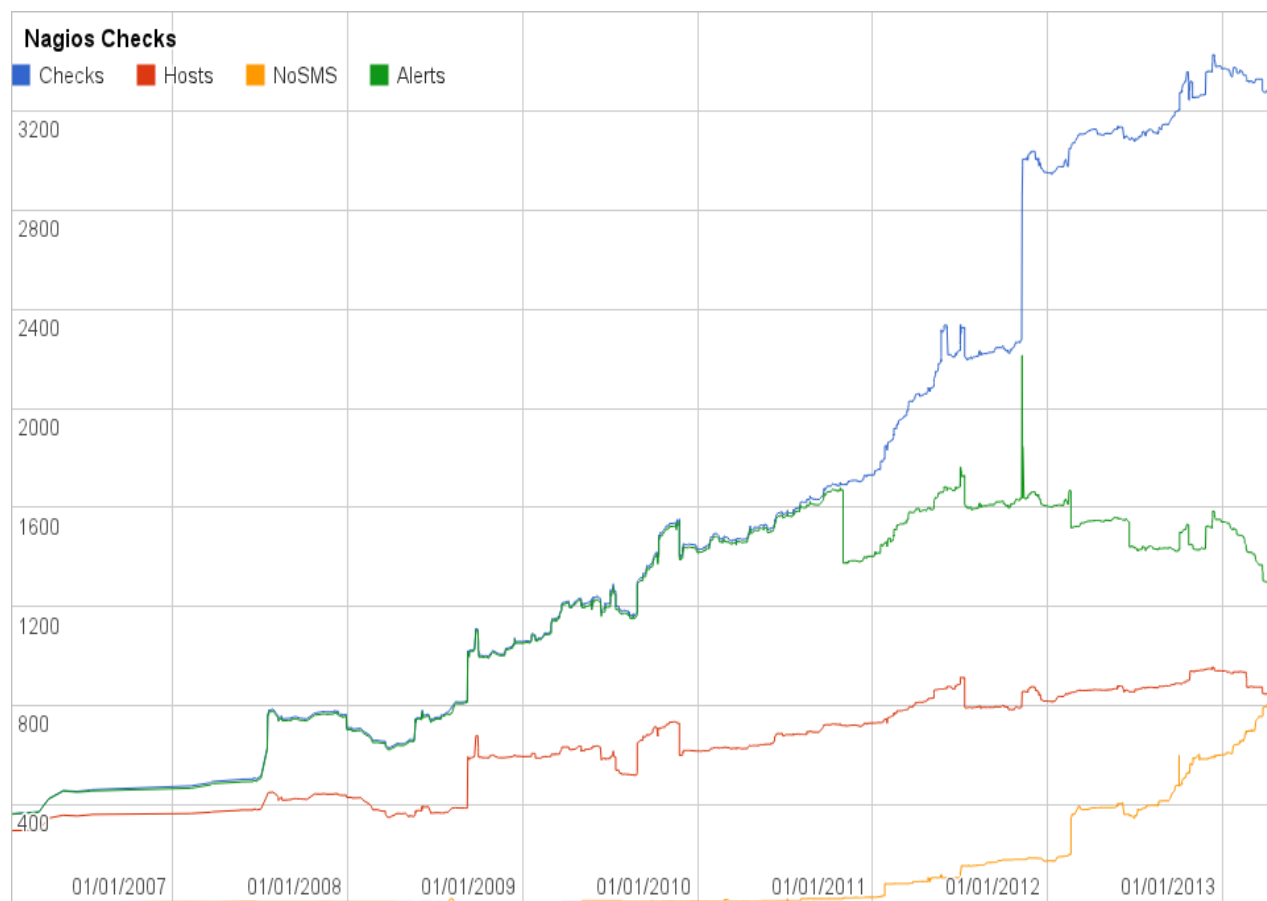


Figure 1. Nagios service checks, hosts being monitored, checks that are configured 'nosms' to not send text alerts, and the total number of alerts going to the team. Note that the total of alerts plus nosms does not total the number of checks since some checks go to other teams.

Poncho: Enabling Smart Administration of Full Private Clouds

Scott Devoid, Narayan Desai

Argonne National Lab

Lorin Hochstein

Nimbus Systems

Abstract

Clouds establish a new division of responsibilities between platform operators and users than have traditionally existed in computing infrastructure. In private clouds, where all participants belong to the same organization, this creates new barriers to effective communication and resource usage. In this paper, we present *poncho*, a tool that implements APIs that enable communication between cloud operators and their users, for the purposes of minimizing impact of administrative operations and load shedding on highly-utilized private clouds.

1. Introduction

With the rise of Amazon EC2 and other public Infrastructure-as-a-Service (IaaS) clouds, organizations are starting to consider private clouds: using a self-service cloud model for managing their computing resources and exposing those resources to internal users.

Open source projects such as OpenStack[19], CloudStack[20], Eucalyptus[21], Ganeti[22], and OpenNebula[23] allow system administrators to build local cloud systems, offering capabilities similar to their public cloud counterparts: the ability to provision computational, storage and networking resources on demand via service APIs. The availability of these APIs provide many advantages over the previous manual approaches: applications can scale elastically as demand rises and falls, users can rapidly prototype on development resources and seamlessly transition to production deployments. On private cloud systems, these activities can occur within a more controlled environment than the public cloud: within the company's private network and without paying a third party (and presumably a profit margin) for resource usage. These systems are quickly becoming a major force in computing infrastructure.

Private clouds face different operational difficulties compared to other large scale systems such as public clouds, traditional server farms, and HPC systems. Private cloud resource management features lag those of public clouds, HPC systems and enterprise infrastructure. Most importantly, resource management capabilities lag other systems, forcing resource underutilization in many cases, and lacking the ability to enforce resource allocation priorities. But the most difficult issue

faced by private cloud operators is the user model. On clouds, users become responsible for some administrative functions, while basic platform management is left to the cloud operators. There is no structured interface between cloud users and cloud operators, resulting in poor coordination between the two. These coordination problems become dire when systems are highly utilized, due to the absence of slack. This is primarily a technical issue, as similar systems are effectively used in large scale compute clusters at similar load. We propose the creation of such an interface, in order to improve effectiveness of private cloud platforms, as well as to ease the operations of these platforms. This effort is the primary contribution of this work.

At Argonne we operate the Magellan system [11], an OpenStack-based private cloud platform dedicated to computational science workloads. Magellan consists of approximately 800 compute nodes of heterogeneous configurations totaling around 7,800 cores, 30 TB of memory and 1.2PB of storage. Resource use is unmetered, but basic system quotas, such as core count, memory, storage capacity, and numbers of VM instances, are enforced. Magellan has been in operation for nearly 30 months as an OpenStack system and for much of this time was the largest deployment of OpenStack in the world. The system supports a large variety of user groups with different workloads, requirements, and expectations.

During this time, we have experienced a variety of issues caused by this lack of communication. Many of these were caused by ill-informed user expectations and high coordination costs. Initially, users drew from their experiences with single physical machines, resulting in lots of independent, unique instances. Even worse, there

was a widespread lack of understanding of the ephemeral storage concept that is widely used in systems. These factors conspired to result in issues where serious user data could be (and occasionally was) lost due to the failure of ephemeral resources. These factors resulted in substantial work in case of resource failures, and caused us to be concerned in cases where service operations required termination of ephemeral virtual machines. In turn, this greatly increased our communication burden when preparing for service operations. This kind of event is representative of a larger class of events where the user support service level should be carefully considered when deploying such a system.

The cloud model for applications, that of horizontally scalable applications with robust fault tolerance, dynamic scalability, and extreme automation, is a poor match for legacy workloads, or some computational science workloads. The former architecture is ideal for cloud operators, as user services are tolerant to failures of underlying instances, while the latter is what many users need in order to achieve their goals. This mismatch is one of the challenges facing private cloud operators. Worse yet, this incongruity is hidden behind the abstractions provided by cloud APIs, limiting the ability of cloud operators and users to effectively collaborate.

This issue can, and must, be alleviated by improving the communication between users and operators. In this paper, we will discuss concrete operational and usability issues caused by this shortcoming, many of which are specific to private clouds. We will present *poncho*¹, a lightweight conduit for API-driven communication via instance annotations, as well as comparing it with comparable facilities in public clouds. This system is currently in the early stages of deployment, with users beginning to incorporate annotations into their workloads.

2. Operational Challenges of Private Clouds

Operationally, private clouds are distinct from public clouds and traditional computing infrastructure (HPC systems and server farms) in several ways. Private clouds are, by their nature, operated by an organization for internal users. While these are similar in many ways to public clouds, this model implies an alignment of

goals between system operators and users that does not exist in the market-based interactions of public clouds. This alignment means that operators and users are invested in deriving the most institutional benefit from private cloud systems, and are expected to collaborate effectively. In many ways, this is analogous to server farms or HPC systems, where incentives are similarly aligned. The cloud user model becomes even more challenging on private clouds; responsibilities are divided responsibilities in a far more complex ways than on traditional infrastructure, and both end users and system operators are expected to collaborate. These factors combine to cause operational challenges in a variety of dimensions. We will discuss these in turn.

2.1 Private Clouds

Private clouds are motivated by a desire to have the best of all possible worlds. Effectively, organizations want the benefits of public clouds in terms of flexibility, availability, and capacity planning, with the greater than 95% utilization rates of large scale HPC systems, and performance of traditional server farms. Also, many organizations want large multi-tenant systems, which enable economies of scale unavailable in unconsolidated infrastructure. Finally, organizations want a cloud where operators and users have aligned incentives, and can collaborate on organizational goals.

As always, the devil is in the details. When building private clouds, several challenges make it difficult to realize this ideal system goal. The state of private cloud software, while improving quickly, lags behind large scale public clouds like AWS. The flexibility of the cloud resource allocation model, where users have unfettered access to resources, requires that different groups perform specialized functions: operators build the cloud platform, while users build services and applications using these resources, and the APIs that encapsulate them. These APIs are insufficient to express the full range of user goals, rather, users specify requests in resource-centric notation. The end results of this approach are a stream of requests that the cloud resource manager attempts to satisfy, with no knowledge of their relative importance, duration, or underlying use case. Because there is no conduit for user intent information, it is difficult for users and operators to coordinate effectively. Moreover, this makes direct collaboration between users and operators, a key benefit of private clouds, considerably more difficult.

¹ Ponchos are useful in circumstances directly following clouds filling.

2.2 The Private Cloud/Openstack Resource Management Model

OpenStack provides APIs to access compute, storage, and networking resources. Resource allocations in OpenStack have no time component; that is, there is no duration. This shortcoming has several important effects, all of which center on resource reclamation. First, resources can't be reclaimed by the system when needed for other work. This limits the ability of the scheduler to implement priority scheduling, as resources are committed to a request once they are awarded, until the user releases them. Second, when the system fills, it becomes effectively useless until resources are released. This disrupts the appearance of elasticity in the system; if users can't request resources and be confident in their requests being satisfied, it causes them to behave pathologically, hoarding resources and so forth. Finally, this model poses serious challenges to the effectiveness goal of private clouds, because the system can't reclaim resources that are being ineffectively used or left idle altogether. This is a distinct goal of private clouds, because resource provider and resource consumer incentives are aligned.

OpenStack only has two methods for implementing resource management policies: request placement, and quotas. Both of these methods are inadequate for multi-tenant systems, where users have competing goals. Resource placement includes methods for selection of resources when new requests arrive. These decisions are *sticky*, that is, they persist until the allocation is terminated, so they aren't useful for implementing policy in steady state operations. Quotas are a component of the solution, and are the only method to implement fairness by default. Because these quotas are static, and are hard quotas, they are a blunt instrument, and can't be used to softly change user behavior.

2.3 Private Cloud User Model and the Role of Platform Operators

One of the major features of private clouds is a reformulation of responsibilities centering on the role of users and platform operators. In the private cloud model, platform operators are responsible for the health of the underlying cloud platform, including API endpoints, and hardware infrastructure, as well as aiming to meet the SLAs for allocated resources. Users are responsible for everything that happens inside of resources. Furthermore, these resources are black boxes; platform operators don't have any visibility into user allocations, or their states. This disconnect is problematic from a variety of perspectives. First, operators are unable to

accurately assess the impact of failures, terminations, and service actions. Second, operators can't determine which resources are in use for tasks important to users, versus lower priority tasks they may be running. Building a channel for communication between users and operators creates an opportunity for explicit collaboration, where only ad-hoc methods previously existed.

3. User/Operator Coordination on Private Clouds

While private clouds are a quickly growing architecture for computing resources, the current state of the art leaves several operational gaps, as described above. In order to address these issues, we propose the addition of two methods for coordination between users and operators. The first of these is an annotation method, whereby users can describe the properties of their VMs. This enables users to communicate requirements and expectations to cloud operators unambiguously. Also, these annotations allow system operators to reclaim resources and take other actions while minimizing user impact. The second component is a notification scheme whereby users are told when their resources are affected by failures, resource contention or administrative operations. Both of these mechanisms are used by the third component, which plans "safe" operations based on user annotations and notifies users as needed. In this section, we will discuss the explicit use cases this work addresses, as well as design and implementation of these features.

3.2 Use Cases

Many private cloud operations are impacted by the lack of good information flows between users and operators, as well as the basic model offered for resource management. We find that users have particular use cases for each of their instances--information that should be communicated to the cloud operators. Operators need to perform a variety of service actions on the resources that comprise the cloud and lack the tools to plan actions while minimizing user impact.

3.2.1 Instance Use Cases

Most of the activity on our system is centered around the following broad use cases. Each of these is impacted by the lack of good communication between operators and users.

Service instances - Service instances implement network accessible services. Often, these services must answer requests immediately, hence have availability

requirements, and have provisioned resources in a high availability configuration. They are managed with the help of auto-scaling software such as AWS CloudFormation or Openstack Heat. Fault tolerance is often implemented at the application layer, which can provide additional flexibility for the platform.

Compute-intensive instances - These instances perform batch-oriented computation or analysis workloads. They are throughput oriented workloads, where the results of computation are needed, but not immediately. Batch queues or task managers usually manage this workload internal to the allocation and can restart failed tasks.

Development instances - These instances have the interactive character of service instances, but none of the HA qualities; users access resources directly for development reasons. These instances are not heavily utilized, as with the previous two use cases, and are only used when the user is active. They may contain unique data in some cases.

Ad-hoc/Bespoke instances - These instances are the wild west. Users treat some instances like physical machines, building custom configurations and running ad-hoc tasks. These instances are the most difficult to support, as they likely contain some unique data, and may have long-running application state that could be lost in event of failures or instance reboots.

3.2.2 Operator Use Cases

Operators need to be able to perform a variety of service actions on the cloud. In both of these cases, user-visible impact must be minimized. This goal is made more difficult by the poor flow of information between users and operators.

Resource Maintenance

Components of the cloud need proactive maintenance, for reasons ranging from software updates and security patches to signs of impending failure. In these situations, operators need to effectively coordinate with users. These processes may be manually or automatically initiated, and depending on the circumstances may be synchronous (in the case of impending failures) or asynchronous (in the case of software updates that may be delayed for a limited time).

Rolling updates fall into this category. These updates need to be performed, but do not necessarily have a short-term deadline. Updates could be performed opportunistically when a resource is free, however, oppor-

tunity decreases as utilization increases. While this approach can result in substantial progress with no user-visible impact, long-running allocations prevent it from being a comprehensive solution; user-visible operations are usually required on system-wide updates.

Load Shedding

In some cases, the cloud needs available resources for new requests, requiring some resource allocations to be terminated. This can occur due to hardware failure, single tenant deadlines, or a lack of fairness in the schedule. Ideally, load shedding minimizes visible impact to user-run services, as well as the loss of local application state. In short, when resource allocations must be terminated, choose wisely. Our initial load shedding goal is to support a basic, synchronous model. More complex policies will follow as future work.

Notifications

A cross-cutting issue with all operator workflows is providing the appropriate notifications to users when actions are taken against resources. Sending an email or opening a service ticket works if an operator manually makes a few service actions during the day. But as service actions are automated, notifications must also become automated.

3.3 Design

The design of poncho is centered around the basic notion that users and operators can coordinate through a combination of resource annotations and system notifications. That is, users and operators agree to mutually beneficial coordination for operations which can potentially cause user-visible outages. These are subtly different from traditional SLAs, where the system operator agrees to provide a particular service level. Rather, in this case, users specify their goals, and the operators provide a best-effort attempt to minimize high impact changes. These goals are approached individually on a tenant by tenant basis, so inter-tenant prioritization doesn't need to be expressed here.

These goals have a few major parts. The first component encodes the impact of service actions on a given instance, and describe conditions where an action will have acceptable impact on the user workload. An example of this is "instance X can be rebooted during the interval between 10PM and 2AM", or "instance Y can be rebooted at any time". The second, closely related part describes how resources should be deallocated, when the system does so. For example, some resources should be snapshotted prior to shutdown, while others can be terminated with no loss of data. A third class of

annotations describe actions the system should take on the user's behalf, such as killing instances after a specified runtime.

The particular annotations we have chosen enable a key resource management capability: load shedding. With the addition of load shedding, more advanced resource management strategies can be implemented, where they were not previously possible. This outcome is a key deliverable of our design; its importance cannot be understated.

The other major part of poncho's architecture is a notification function. Users can register to be notified when service actions are performed. These notifications describe the resources affected, the action taken, and a basic reason for the action. For example, a notification might tell a user that "instance Z was terminated because of a load shedding event". This would signal that requests to re-instantiate the instance would likely fail. Alternatively, a notification like "instance Z was terminated due to failure" would signal that capacity is likely available for a replacement allocation request. Notifications are delivered on a best effort basis, with a limited number of immediate retries, but no guarantee of reliable delivery. As most of this information is available through default APIs in an explicit way, applications can poll as a fallback.

3.3.1 Annotation API

We have modeled instance annotations as a series of key/value pairs, stored as instance metadata via the pre-existing mechanism in OpenStack. [2] These values are described in the table below. Examples of common use cases are show in the following examples section.

Table 2 : Instance annotations, metadata

Key Name	Description
reboot_when	Semicolon delimited list of conditions, see Table 3.
terminate_when	Semicolon delimited list of conditions, see Table 3.
snapshot_on_terminate	Boolean; create a snapshot of the instance before terminating.
notify_url	URL of service receiving event notifications.
ha_group_id	Tenant-unique ID of service HA group.
ha_group_min	Minimum number of instances within the HA group.

Table 3 : Conditional grammar

Condition example	Description
"MinRuntime(duration)"	True if the instance has been running for the specified duration.
"Notified(interval)"	True if the interval has elapsed since a scheduled event notification was sent.
"TimeOfDay(start, stop, tz)"	True if the time of day is between start and stop with the optional time zone offset from UTC. Example: "TimeOfDay(22:00, 02:00, -05:00)".

These attributes specify user goals pertaining to instance reboots and termination, as well as whether instances should be snapshotted upon termination. Users can specify a notification URL where events are submitted, and a tenant-specific high availability group ID. The priority attribute is used to choose between instances when load shedding occurs. If a tenant is chosen for load shedding, and multiple instances are flagged a terminatable, these instances are ordered in ascending order by priority, and the first instance(s) in the list are selected for termination. Priority settings of one tenant do not affect which instances are shed in another tenant.

The high availability group annotations provide a limited set of features: they ensure that cloud operators do not load shed instances that are part of that group and leave it with less than the minimum number of instances allowed. In this implementation the user is still responsible for determining scale-up needs and identifying an HA group failures that occur outside of planned operations.

The conditional grammar terms shown in Table 3 describe when terminate or reboot actions have acceptable consequences to the user. If multiple predicates are specified, all must be satisfied for the operation to be deemed safe. Note that this condition is merely advisory; failures or other events may result in resource outages causing user impacting service outages regardless of these specifications. This difference is the major distinction between these specifications and SLAs.

3.3.2 Notification API

The primary goal of the notification API is to inform user about system actions that impact their instances. By annotating the instance with a “notify_url” tag, the user can specify a URL that listens for events from poncho. Events are sent as JSON encoded HTTP POST requests to the “notify_url”. All events contain the following basic attributes:

- “timestamp” : A timestamp for the event
- “type” : An event type, from a fixed list.
- “description” : A descriptive explanation of why this event is happening.

Specific event types contain additional attributes, listed in Table 4.

Table 4. Description of notification event types

Event Type	Description and supplemental information
reboot_scheduled	A reboot has been scheduled. Includes the instance ID, name and reboot time.
rebooting	The instance is now rebooting. Includes the instance ID, name.
terminate_scheduled	The instance has been scheduled to be terminated. Includes the instance ID, name and a termination time.
terminating	The instance is now being terminated. Includes the instance ID and name.
terminated	The instance was terminated at some point in the past. This notification is used for service failures where the instance cannot be recovered. Includes the instance ID and name.
snapshot_created	A snapshot of the instance has been created. Includes the instance ID, instance name and the ID of the created snapshot.
ha_group_degraded	The HA group for this instance no longer has the minimum number of instances. Includes the HA group ID and a list of instance IDs for instances still active within that group. Sent once per HA group.
ha_group_healthy	The HA group for this instance has transition from degraded to healthy. Includes the HA group ID and the list of instances active within the group. Sent once per HA group.
shed_load_request	A request by the operators to the tenant to deallocate instances if possible. This is sent out once for every unique notification URL within the tenant.

Currently instances default to no notification URL. We have implemented an optional configuration of Poncho that formats messages for these instances as an email to the instance owner. For the HA group and shed-load events, messages are sent as emails to the tenant administrators.

User-written notification agents are fairly simple. A server responds to the HTTP endpoint registered as a notification URL, and takes appropriate actions. While simple notification agents are fairly general, we have found that most tenants want custom policies depending on their needs.

3.4 Implementation

We implemented poncho in three parts. The first is a set of scripts that provide a user-centric command line interface to annotate nodes. The second is a notification library that is used by administrative scripts to notify userspace agents upon administrative action. The third is a set of administrative scripts that can be run interactively or periodically to shed load, service nodes, or kill time limited tasks. This final component is run periodi-

cally in our initial prototype. The primary goal of this prototype is to gain some experiences coordinating with users in a productive fashion, so the system itself is deliberately simplistic until we validate our basic model.

Our initial implementation of poncho is intended to function as a force multiplier, whereby administrators

and users perform roughly similar sorts of tasks with the aid of scripts that streamline these processes. Operators gain the ability to perform some service actions in an automated fashion, and begin to understand the impact of service options. Users gain the ability to submit allocation requests for fixed duration, with automatic termination, as well as the ability to communicate information about their workloads, like the impact of instance outages.

Poncho is an open-source Python application, leveraging existing OpenStack Python APIs and is compatible with any Openstack deployment running Essex or newer releases. It is available on Github [12].

While we hope to integrate similar functionality into Openstack, this version has been implemented in a minimally invasive fashion. Our goal in this effort is to gain sufficient experience to develop a comprehensive model for user/operator interactions. Once we have some confidence in our model, we plan to develop an Openstack blueprint and an implementation suitable for integration into Openstack itself.

3.5 Example Use Cases

For instances that have no annotations, a default annotation is assumed which meets most users expectations for cloud instances:

```
{ "terminate_when" : false, "reboot_when" : true }
```

This annotation declares instance reboots to be safe at any time, but terminations to be deemed unsafe at any time.

Running an instance for development work is a common use case on Magellan. For this case, we define a minimum runtime of twelve hours, a full day of work, before the instance can be terminated; we also enable automatic snapshotting since the user may have important work that needs to be saved. Our conservative policy is for tenants to delete unnecessary snapshots.

```
{ "terminate_when" : "MinRuntime(12h)", snapshot_on_terminate : true }
```

For workloads that are throughput oriented, there are a number of annotation configurations that might work. The following annotation ensures that a minimum number of instances are working for the HA group, that the user is notified one hour before any scheduled events and that this instance is only considered after instances in the same tenant with a lower priority score:

```
{ "terminate_when" : "Notified(1h)", "ha_group_id" : 12, "ha_group_min" : 5, priority : 10, "notify_url" : "http://example.com/notifications" }
```

4. Experiences and Discussion

An initial version of poncho has been deployed to users on Magellan. Many of our tenants are invested in helping us to develop the user model and resource management capabilities, because they notice the lack of communication, and feel they are using resources inefficiently. Initial user responses have been enthusiastic.

At this point, our two largest tenants have begun to use these interfaces. One of these tenants has a throughput dominated workload, and had previously communicated this to us in an ad-hoc manner. In effect, the interfaces provided by this system formalize a manual arrangement. Another of our major tenants has a development heavy workload, and had been looking for a system that reaped old instances after taking snapshots for several months. A third tenant, with a workload that consists of a combination of development and throughput-oriented instances has also agreed to begin using these interfaces as well. At this point, we have only tested poncho's functionality in an artificial setting; we have not needed to shed load or service poncho-mediated resources yet; the system is fairly reliable, and all instances are not yet tagged.

Our initial experiences with users have shown two basic models. Users with throughput oriented workloads are able to integrate these methods into their workflows relatively easily, as all of their resources are started up in a uniform way. Interactive users, largely instantiating development instances, start their instances in a variety of different ways, making uniform adoption considerably more difficult. These latter kinds of instances consume resources in a bursty fashion, while occupying resources consistently. In our experiences, tenants want to set custom policies for their development instances. This approach is similar in philosophy to the one taken by Netflix's Janitor Monkey, and consolidates policy at the tenant level, not with either the system or individual users.

It remains an open question how broad adoption of these APIs will be across tenants on our system. For this reason, it was critical to set a reasonable default set of annotations for instances. Once clear conservative option is to define both instance reboots and terminations as invasive. Another more flexible option allows re-

boots with 24 hour calendar notice emailed to users while still deeming terminations as invasive. We have chosen this latter option, as it gives users some incentive to learn these APIs and put them into use if they have a sensitive workload.

With the addition of load-shedding capabilities, we enable Openstack to implement a range of scheduling algorithms familiar from public clouds and HPC systems. This core capability is the fundamental infrastructure for AWS spot instances, and HPC system scavenger queues. We plan to explore these options for improving system productivity.

Clouds put operators/system administrators into the role of building API driven services for their organizations. These new services implicitly include a collaborative function with users, with a division of responsibilities (which is familiar to administrators) and an abstraction barrier, which is new. In our view, it is critical that cloud operators be proactive, and help to design effective coordination facilities to enable users to use resources effectively, both in throughput-oriented computational workloads and availability-oriented service workloads. APIs, which provide services to users, can just as easily be used to provide services to operators. With this sort of approach, traditionally expensive problems can be simply solved. Solutions of this kind are critical if private clouds are to grow to their full potential.

5. Background and Related Work

The problems of effective communication with users to enable efficient resource management, including load-shedding, are old ones in system management. In HPC systems, resource allocations are explicitly annotated with a maximum runtime. HPC schedulers, such as Maui[17], and Slurm[18] can use these annotations to implement scheduling algorithms such as conservative backfill[25]. The availability of maximum runtimes also enable deterministic draining of resources, a luxury unavailable on private clouds due to the private cloud resource allocation model. In [24], the authors explore instituting explicit resource leases on top of the cloud resource allocation model.

Public clouds have some features that enable effective coordination between platform operators and cloud users. Amazon's Spot Instances[15] are a prime example of that, and is specifically built on top of load-shedding techniques.

AWS and Rackspace both have an API specifically for coordinating planned outages. Instance and hypervisor reboots are scheduled and that schedule is queryable by the user. In some cases users may elect to reboot instances at a time of their choosing before the scheduled maintenance window.[1,3] This enables users to perform the required upgrades in a controlled fashion, e.g. when personnel are available to diagnose and fix unexpected issues with the upgrade.

Support for high availability service groups are widely supported; AWS and Microsoft Azure include mechanisms to build such services. [1,7] In private cloud software stacks, Openstack (via Heat[16]) and Eucalyptus both support similar mechanisms.

Several systems provide auto-scaling functionality, which could be configured to receive events from poncho. AWS includes integrated auto-scaling services, as does Azure. Heat provides related functionality for the Openstack ecosystem. Some of these systems use VM profiling tools to identify applications with high CPU or memory load to be scaled.

Public cloud operators manage user expectations with SLAs. This approach is quite effective in conjunction with variable pricing across service classes. Our approach is slightly different, using annotations to signal user desires, as opposed to making guarantees to the users.

Rightscale[26] and Cycle Computing[27] are third party resource management environments that implement advanced policies on top of public and private clouds for service and throughput-oriented workloads, respectively.

In [8], the authors propose a strategy of improving private cloud utilization with high-throughput Condor tasks, which can implicitly be terminated at any time. This strategy, similar to Amazon's spot instance strategy, is less flexible than coordination solution presented in this paper, as it solely addresses the utilization problem, not the more general resource evacuation and load shedding problems.

Netflix's Simian Army[13] contains some resource management features that we intentionally designed into poncho. While most of the Simian Army application is concerned with testing the fault tolerance of Netflix's video streaming services, the Janitor Monkey application identifies idle development instances and terminates them after warning the owner. [14]

6. Conclusions

In this paper, we have presented the design and implementation of poncho, a tool that enables better communication between private cloud operators and their users, as well as early experiences with the tool. Our initial implementation of poncho is relatively simplistic, and primarily aims to validate our model for API-driven coordination between users and cloud operators. Poncho has been deployed to users, and initial feedback has been positive, suggesting users are willing to make use of such interfaces if it makes their lives easier or enables more efficient use of computing resources.

One goal in writing this paper was to begin a community discussion of this communication breakdown. As adoption of private clouds grows, these issues will grow more serious, particularly as these systems become more saturated. As a largely non-technical issue, we believe that broad experimentation, on real users, is the best way to develop effective solutions. Moreover, it is critical that system administrators, as they become private cloud operators, remain cognisant of these issues, and strive to minimize their impact.

Finally, as service-oriented computing infrastructure, like private clouds, becomes widespread, operators (and system administrators) will increasingly find their users hidden away behind abstraction barriers, from virtual machines to PaaS software and the like. Both building effective collaborative models with users, and designing APIs that are efficient and productive, are critical tasks that system administrators are uniquely equipped to address within their organizations.

7. Future Work

This work is a set of first steps toward improved communication between users and operators on Openstack private clouds. The prototype described here is a simple implementation of such a conduit, no doubt it will be refined or even redesigned as users develop more sophisticated requirements, driven by their application workloads.

One of the critical pieces of infrastructure provided by this system is a mechanism that can be used for load shedding, as well as a way to communicate with users when this action is required. As a building block, load shedding enables a whole host of more advanced resource management capabilities, like spot instances, advanced reservations, and fairshare scheduling. After our initial assessment of this coordination model is

complete, we plan to build an active implementation, that can directly implement these features.

Notifications, particularly the explicit load-shedding request, enable the creation of hierarchical cooperative resource managers, which is probably the best path forward for integration with traditional resource managers.

References

- [1] **Amazon EC2 Maintenance Help**, <http://aws.amazon.com/maintenance-help/> on 4/30/2013.
- [2] **OpenStack API Documentation** <http://api.openstack.org/api-ref.html>
- [3] **Preparing for a Cloud Server Migration**, retrieved from http://www.rackspace.com/knowledge_center/article/preparing-for-a-cloud-server-migration on 4/30/13.
- [4] **Provisioning Policies for Elastic Computing Environments**, Marshall, P., Tufo, H., Keahey, K. *Proceedings of the 9th High-Performance Grid and Cloud Computing Workshop and the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Shanghai, China. May 2012.
- [5] **A Comparison and Critique of Eucalyptus, OpenNebula and Nimbus**, Sempolinski, P.; Thain, D. *IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, 2010, pp.417-426, Nov. 30 2010-Dec. 3 2010 <http://www3.nd.edu/~ccl/research/papers/psempoli-cloudcom.pdf>
- [6] **Nova API Feature Comparison**, retrieved from <https://wiki.openstack.org/wiki/Nova/APIFeatureComparison> on 4/30/13.
- [7] **Windows Azure Execution Models**, retrieved from <http://www.windowsazure.com/en-us/develop/net/fundamentals/compute/> on 4/30/13.
- [8] **Improving Utilization of Infrastructure Clouds**, Marshall, P., Keahey K., Freeman, T. *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2011)*, Newport Beach, CA. May 2011.
- [9] **Manage the Availability of Virtual Machines**, <http://www.windowsazure.com/en-us/manage/linux/common-tasks/manage-vm-availability/> retrieved on 30 April 2013.
- [10] **Troubleshooting in Windows Azure**, <http://www.windowsazure.com/en-us/manage/linux/best-practices/troubleshooting/> retrieved on 30 April 2013.
- [11] **Magellan: Experiences from a Science Cloud**, Lavanya Ramakrishnan, Piotr T. Zbiegel, Scott Camp-

- bell, Rick Bradshaw, Richard Shane Canon, Susan Coghlan, Iwona Sakrejda, Narayan Desai, Tina Declerck, Anping Liu, *Proceedings of the 2nd International Workshop on Scientific Cloud Computing*, ACM ScienceCloud '11, 2011
- [12] **Poncho Github Repository**, retrieved from <https://github.com/magellanccloud/poncho> on 4/30/13
- [13] **The Netflix Simian Army**, Y. Izrailevsky, A. Tseitlin, *The Netflix Tech Blog*, 19 July 2011, <http://techblog.netflix.com/2011/07/netflix-simian-army.html>, retrived on 30 April 2013
- [14] **Janitor Monkey: Keeping the Cloud Tidy and Clean**, M. Fu, C. Bennett, *The Netflix Tech Blog*, <http://techblog.netflix.com/2013/01/janitor-monkey-keeping-cloud-tidy-and.html>, retrived on 30 April 2013.
- [15] **Amazon EC2 Spot Instances**, <http://aws.amazon.com/ec2/spot-instances/>, 30 April 2013.
- [16] **Heat: A Template based orchestration engine for OpenStack**, S. Dake, *OpenStack Summit, San Diego*, San Diego CA, October 2012, <http://www.openstack.org/summit/san-diego-2012/openstack-summit-sessions/presentation/heat-a-template-based-orchestration-engine-for-openstack>, retrived on 30 April 2013.
- [17] **Core Algorithms of the Maui Scheduler**, David Jackson, Quinn Snell, and Mark Clement, *Proceedings of Job Scheduling Strategies for Parallel Processors (JSSPP01)*, 2001.
- [18] **SLURM: Simple Linux Utility for Resource Management**, A. Yoo, M. Jette, and M. Grondona, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 44-60, Springer-Verlag, 2003.
- [19] **OpenStack: Open source software for building public and private clouds**, <http://www.openstack.org/> retrived on 30 April 2013.
- [20] **Apache CloudStack: Open source cloud computing**, <http://cloudstack.apache.org/> retrived on 30 April 2013.
- [21] **The Eucalyptus Open-source Cloud-computing System**. D. Nurmi, R. Wolski, C. Grzegorzczuk, G. Obertelli, S. Soman, L. Youseff, and D.i Zagorodnov. In *Cluster Computing and the Grid*, 2009. *CCGRID'09. 9th IEEE/ACM International Symposium on*, pp. 124-131. IEEE, 2009.
- [22] **Ganeti: Cluster-based virtualization management software**, <https://code.google.com/p/ganeti/> retrived on 30 April 2013.
- [23] **OpenNebula: The open source virtual machine manager for cluster computing**, J. Fontán, T. Vázquez, L. Gonzalez, R. S. Montero, and I. M. Llorente, *Open Source Grid and Cluster Software Conference*. 2008.
- [24] **Capacity leasing in cloud systems using the opennebula engine**. B. Sotomayor, R.S. Montero, I.M. Llorente, and I. Foster. "In *Workshop on Cloud Computing and its Applications*, 2008.
- [25] **The ANL/IBM SP Scheduling System**. D. Lifka. In *Job Scheduling Strategies for Parallel Processing*, pp. 295-303. Springer Berlin Heidelberg, 1995.
- [26] **Rightscale Cloud Management**, <http://www.rightscale.com>, retrieved 20 August, 2013.
- [27] **Cycle Computing**, <http://www.cyclecomputing.com>, retrieved 20 August, 2013.

Making Problem Diagnosis Work for Large-Scale, Production Storage Systems

Michael P. Kasick, Priya Narasimhan
Electrical & Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA 15213-3890
{mkasick, priyan}@andrew.cmu.edu

Kevin Harms
Argonne Leadership Computing Facility
Argonne National Laboratory
Argonne, IL 60439
harms@alcf.anl.gov

Abstract

Intrepid has a very-large, production GPFS storage system consisting of 128 file servers, 32 storage controllers, 1152 disk arrays, and 11,520 total disks. In such a large system, performance problems are both inevitable and difficult to troubleshoot. We present our experiences, of taking an automated problem diagnosis approach from proof-of-concept on a 12-server test-bench parallel-file-system cluster, and making it work on Intrepid's storage system. We also present a 15-month case study, of problems observed from the analysis of 624 GB of Intrepid's instrumentation data, in which we diagnose a variety of performance-related storage-system problems, in a matter of hours, as compared to the days or longer with manual approaches.

Tags: problem diagnosis, storage systems, infrastructure, case study.

1 Introduction

Identifying and diagnosing problems, especially performance problems, is a difficult task in large-scale storage systems. These systems are comprised of many components: tens of storage controllers, hundreds of file servers, thousands of disk arrays, and tens-of-thousands of disks. Within high-performance computing (HPC), storage often makes use of parallel file systems, which are designed to utilize and exploit parallelism across all of these components to provide very high-bandwidth concurrent I/O.

An interesting class of problems in these systems is hardware component faults. Due to redundancy, generally component faults and failures manifest in degraded performance. Due to careful balancing of the number of components and their connections, the degraded performance of even a single hardware component may be observed throughout an entire parallel file system, which makes problem localization difficult.

At present, storage system problems are observed and diagnosed through independent monitoring agents that exist within the individual components of a system, e.g.,

disks (via S.M.A.R.T. [11]), storage controllers, and file servers. However, because these agents act independently, there is a lack of understanding how a specific problem affects overall performance, and thus it is unclear whether a corrective action is immediately necessary. Where the underlying problem is the misconfiguration of a specific component, an independent monitoring agent may not even be aware that a problem exists.

Over the past few years, we have been exploring the use of peer-comparison techniques to identify, locate, and diagnose performance problems in parallel file systems. By understanding how individual components may exhibit differences (asymmetries) in their performance relative to peers, and, based on the presence of these asymmetries, we have been able to identify the specific components responsible for overall degradation in system performance.

Our previous work. As described in [17], we automatically diagnosed performance problems in parallel file systems (in PVFS and Lustre) by analyzing black-box, OS-level performance metrics on every file server. We demonstrated a proof-of-concept implementation of our peer-comparison algorithm by injecting problems during runs of synthetic workloads (dd, IOzone, or PostMark) on a controlled, laboratory test-bench storage cluster of up to 12 file servers. While this prototype demonstrated that peer comparison is a good foundation for diagnosing problems in parallel file systems, it did not attempt to tackle the practical challenges of diagnosis in large-scale, real-world production systems.

Contributions. In this paper, we seek to adapt our previous approach for the primary high-speed storage system of Intrepid, a 40-rack Blue Gene/P supercomputer at Argonne National Laboratory [21], shown in Figure 1. In doing so, we tackle the practical issues in making problem diagnosis work in large-scale environment, and we also evaluate our approach through a 15-month case study of practical problems that we observe and identify within Intrepid's storage system.

The contributions of this paper are:



Figure 1: Intrepid, consists of 40 Blue Gene/P racks [2].

- Outlining the pragmatic challenges of making problem diagnosis work in large-scale storage systems.
- Adapting our proof-of-concept diagnosis approach, from its initial target of a 12-server experimental cluster, to a 9,000-component, production environment consisting of file servers, storage controllers, disk arrays, attachments, etc.
- Evaluating a case study of problems observed in Intrepid’s storage system, including those that were previously unknown to system operators.

We organize the rest of this paper as follows. We start with a description of our approach, as it was originally conceived, to work in a small-scale laboratory environment (see § 2). We then discuss the challenges of taking the initial algorithm from its origin in a limited, controllable test-bench environment, and making it effective in a noisy, 9,000-component production system (see § 3). Finally, we present the new version of our algorithm that works in this environment, and evaluate its capability to diagnose real-world problems in Intrepid’s storage system (see § 4).

2 In the Beginning ...

The defining property of parallel file systems is that they parallelize accesses to even a single file, by striping its data across many, if not all, file servers and logical storage units (LUNs) within a storage system. By striping data, parallel file systems maintain similar I/O loads across system components (peers) for all non-pathological client workloads. In our previous work [17], we hypothesized that the statistical trend of I/O loads, as reflected in OS-level performance metrics, should (i) exhibit symmetry across fault-free components, and (ii) exhibit asymmetries across faulty components. Figure 2 illustrates the intuition behind our hypothesis; the injection of a rogue workload on a spindle shared with a PVFS LUN results in a throughput asymmetry between the faulty and fault-free LUNs, where previously throughput

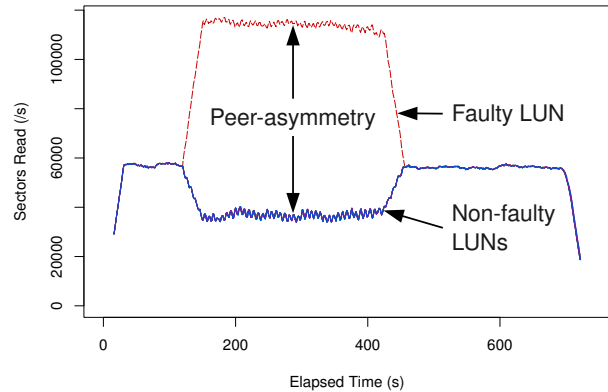


Figure 2: Asymmetry in throughput for an injected fault; provides intuition behind the peer-comparison approach that serves as a good foundation for our diagnosis [17].

was similar across them.

In the context of our diagnosis approach, *peers* represent components of the same type or functionality that are expected to exhibit similar request patterns. By capturing performance metrics at each peer, and comparing these metrics across peers to locate asymmetries (*peer-comparison*), we expect to be able to identify and localize faults to the culprit peer(s).

To validate our hypothesis, we explored a peer-comparison-based approach to automatically diagnose performance problems through a set of experiments on controlled PVFS and Lustre test-bench clusters [17]. In summary, these experiments are characterized by:

- Black-box instrumentation consisting of samples of OS-level storage and network performance metrics.
- Two PVFS and two Lustre test-bench clusters, containing 10 clients and 10 file servers, or 6 clients and 12 file servers, for four clusters in total.
- File servers each with a single, locally-attached storage disk.
- Experiments, both fault-free and fault-injected, approximately 600 seconds in duration, where each client runs the same file system benchmark (dd, IOzone, or PostMark) as a synthetic workload.
- Fault-injection of approximately 300 seconds in duration, consisting of two storage-related, and two network-related performance problems.
- A peer-comparison diagnosis algorithm that is able to locate the faulty server (for fault-injection experiments), and determine which of the four injected problems is present.

In [17], we evaluate the accuracy of our diagnosis with true- and false-positive rates for diagnosing the correct faulty server and fault type (if an injected fault exists). Although our initial diagnosis algorithm exhibited weaknesses that contributed to misdiagnoses in our re-

sults, overall our test-bench experiments demonstrated that peer-comparison is a viable method for performing problem diagnosis, with low instrumentation overhead, in parallel file systems.

3 Taking it to the Field

Following the promising success of our PVFS and Lustre test-bench experiments, we sought to validate our diagnosis approach on Intrepid's primary high-speed GPFS file system (we describe GPFS in § 3.2 and Intrepid's architecture in § 3.3).

In doing so, we identified a set of *new challenges* that our diagnosis approach would have to handle:

1. A large-scale, multi-tier storage system where problems can manifest on file servers, storage attachments, storage controllers, and individual LUNs.
2. Heterogeneous workloads of unknown behavior and unplanned hardware-component faults, both of which are outside of our control, that we observe and characterize as they happen.
3. The presence of system upgrades, e.g., addition of storage units that see proportionally higher loads (non-peer behavior) as the system seeks to balance resource utilization.
4. The need for continuous, 24/7 instrumentation and analysis.
5. Redundant links and components, which also exhibit changes in load (as compared to peers) when faults are present, even though the components themselves are operating appropriately.
6. The presence of occasional, transient performance asymmetries that are not conclusively attributable to any underlying problem or misbehavior.

3.1 Addressing these New Challenges

While problem diagnosis in Intrepid's storage system is based on the same fundamental peer-comparison process we developed during our test-bench experiments, these new challenges still require us to adapt our approach at every level: by expanding the system model, revisiting our instrumentation, and improving our diagnosis algorithm. Here we map our list of challenges to the subsequent sections of the paper where we address them.

Challenge #2. Tolerating heterogeneous workloads and unplanned faults are inherent features of our peer-comparison approach to problem diagnosis. We assume that client workloads exhibit similar request patterns across all storage components, which is a feature provided by parallel file system data striping for all but pathological cases. We also assume that at least half of the storage components (within a peer group) exhibit fault-free behavior. As long as these assumptions hold, our peer-comparison approach can already distinguish

problems from legitimate workloads.

Challenges #1, #3, and #5. Unlike our test-bench, which consisted of a single storage component type (PVFS or Lustre file server with a local storage disk), Intrepid's storage system consists of multiple component types (file servers, storage controllers, disk arrays, attachments, etc.), that may be amended or upgraded over time, and that serve in redundant capacities. Thus, we are required to adapt our system model to tolerate each of these features. Since we collect instrumentation data on file servers (see § 4.1), we use LUN-server attachments as our fundamental component for analysis. With knowledge of GPFS's prioritization of attachments for shared storage (see § 3.3.2), we handle redundant components (challenge #5) by separating attachments into different priority groups that are separately analyzed. We handle upgrades (challenge #3) similarly, separating components into different sets based on the time at which they're added to the system, and perform diagnosis separately within each upgrade set (see § 3.3.1). Furthermore, by knowing which attachments are affected at the same time, along with the storage system topology (see § 3.3), we can infer the most likely tier and component affected by a problem (challenge #1).

Challenge #4. As in [17] we use `sadc` to collect performance metrics (see § 4.1). To make our use of `sadc` amenable to continuous instrumentation, we also use a custom daemon, `cycle`, to rotate `sadc`'s activity files once a day (see § 4.1.1). This enables us to perform analysis on the previous day's activity files while `sadc` generates new files for the next day.

Challenge #6. Transient performance asymmetries are far more common during the continuous operation of large-scale storage systems, as compared to our short test-bench experiments. Treatment of these transient asymmetries requires altering the focus of our analysis efforts and enhancing our diagnosis algorithm to use persistence ordering (see § 4.3).

3.2 Background: GPFS Clusters

The General Parallel File System (GPFS) [27] is a cluster and parallel file system used for both high-performance computing and network storage applications. A GPFS storage cluster consists of multiple file servers that are accessed by one or more client nodes, as illustrated for Intrepid in Figure 3. For large I/O operations, clients issue simultaneous requests across a local area network (e.g., Ethernet, Myrinet, etc.) to each file server. To facilitate storage, file servers may store data on local (e.g., SATA) disks, however, in most clusters I/O requests are further forwarded to dedicated storage controllers, either via direct attachments (e.g., Fibre Channel, InfiniBand) or over a storage area network.

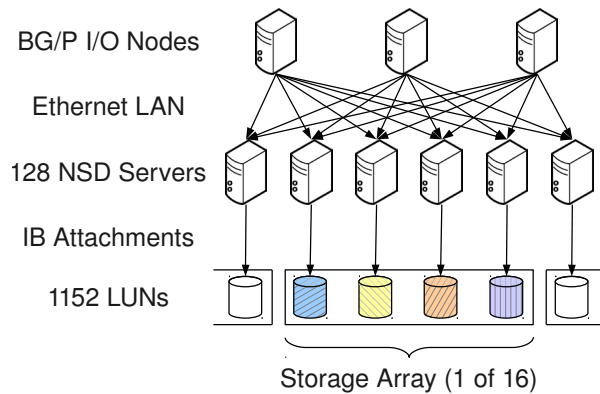


Figure 3: Intrepid's storage system architecture.

Storage controllers expose block-addressable logical storage units (LUNs) to file servers and store file system content. As shown in Figure 4, each LUN consists of a redundant disk array. Controllers expose different subsets of LUNs to each of its attached the file servers. Usually LUNs are mapped so each LUN primarily serves I/O for one (primary) file server, while also allowing redundant access from other (secondary) file servers. This enables LUNs to remain accessible to clients in the event that a small-number of file servers go offline. Controllers themselves may also be redundant (e.g., coupled “A” and “B” controllers) so that LUNs remain accessible to secondary file servers in the event of a controller failure.

The defining property of parallel file systems, including GPFS, is that they parallelize accesses to even a single file, by striping its data across many (and in a common configuration, across all) file servers and LUNs. For example, when performing large, sequential I/O, clients may issue requests, corresponding to adjacent stripe segments, round-robin to each LUN in the cluster. LUNs are mapped to file servers so that these requests are striped to each file server, parallelizing access across the LAN, and further striped across the primary LUNs attached to file servers, parallelizing access across storage attachments.

The parallelization introduced by the file system, even for sequential writes to a single file, ensures that non-pessimistic workloads exhibit equal loads across the cluster, which in turn, should be met with balanced performance. Effectively, just as with other parallel file systems, GPFS exhibits the characteristics that make peer-comparison a viable approach for problem diagnosis. Thus, when “hot spots” and performance imbalances arise in a cluster, we hypothesize them to be indicative of a performance problem. Furthermore, by instrumenting each file server in the cluster, we can observe the performance of file servers, storage controllers, and LUNs, from multiple perspectives, which enables us to localize problems to the components of the cluster where performance imbalance is most significant.

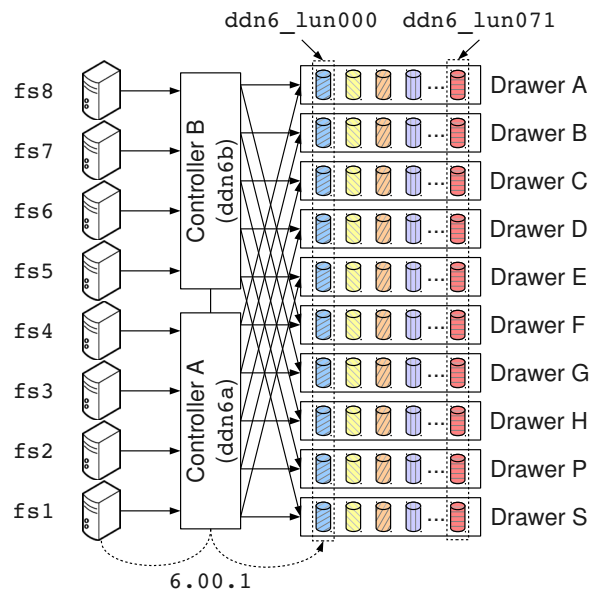


Figure 4: Storage array subarchitecture, e.g., ddn6.

3.3 Intrepid's Storage System

The target of our case study is Intrepid's primary storage system, a GPFS file system that consists of 128 Network Shared Disk (NSD) servers (*fs1* through *fs128*) and 16 DataDirect Networks S2A9900 storage arrays, each with two storage controllers [21]. As illustrated in Figure 4, each storage array exports 72 LUNs (*ddn6_lun000* through *ddn21_lun071*) for Intrepid's GPFS file system, yielding a 4.5 PB file system comprised from 1152 LUNs and 11,520 total disks. At this size, this storage system demands a diagnosis approach with scalable data volume and an algorithm efficient enough to perform analysis in real-time with modest hardware. In addition, because our focus is on techniques that are amenable to such production environments, we require an approach with a low instrumentation overhead.

3.3.1 System Expansion

Of the 72 LUNs exported by each storage array, 48 were part of the original storage system deployment, while the other 24 were added concurrently with the start of our instrumentation to expand the system's capacity. Since the 24 LUNs added in each storage array (384 LUNs total) were initially empty, they observe fewer reads and more writes, and thus, exhibit non-peer behavior compared to the original 48 LUNs in each array (768 LUNs total). As our peer-comparison diagnosis approach performs best on LUNs with similar workloads, we partition Intrepid into “old” and “new” LUN sets, consisting of 768 and 384 LUNs respectively, and perform our diagnosis separately within each set.

3.3.2 Shared Storage

Each Intrepid LUN is redundantly attached to eight GPFS file servers with a prioritized server ordering defined in a system-wide configuration. We denote these LUN-server attachments with the convention `controller.lun.server`, e.g., `6.00.1` or `21.71.128`.

GPFS clients, when accessing a LUN, will route all I/O requests through the highest-priority, presently-available server defined for that LUN. Thus, when all servers are online, client I/O requests route through the primary server defined for a given LUN. If the primary server is unavailable, requests route through the LUN’s secondary, tertiary, etc., servers based on those servers availability.

Since redundant attachments do not have equal priority for a given LUN, this effectively creates eight system-wide priority groups consisting of equal-priority LUN-server attachments, i.e., the first priority-group consists of all primary LUN-server attachments, the second priority-group consists of all the secondary LUN-server attachments, etc. Combined with the “system expansion” division, the total of 9216 LUN-server attachments ($1152 \text{ LUNs} \times 8 \text{ redundantly attached servers}$) must be analyzed in 16 different peer groups ($8 \text{ priority groups} \times 2 \text{ for “old” vs. “new” LUNs}$) in total.

4 Making it Work for Intrepid

As we apply our problem-diagnosis approach to large storage systems like Intrepid’s, our primary objective is to locate the most problematic LUNs (specifically LUN-server attachments that we refer to as “LUNs” henceforth) in the storage system, which in turn, reflect the location of faults with greatest performance impact. This process consists of three stages:

Instrumentation, where we collect performance metrics for every LUN (see § 4.1);

Anomaly Detection, where we identify LUNs that exhibit anomalous behavior for a specific window of time (see § 4.2);

Persistence Ordering, where we locate the most problematic components by their persistent impact on overall performance (see § 4.3).

4.1 Instrumentation

For our problem diagnosis, we gather and analyze OS-level storage performance metrics, without requiring any modifications to the file system, the applications or the OS. As we are principally concerned with problems that manifest at the layer of NSD Servers and below (see Figure 3), the metrics that we gather and utilize consist of the storage-metric subset of those collected in our previous work [17].

Metric	Significance
<code>tps</code>	Number of I/O (read and write) requests made to (a specific) LUN per second.
<code>rd_sec</code>	Number of sectors read from the LUN per second.
<code>wr_sec</code>	Number of sectors written to the LUN per second.
<code>avgrq-sz</code>	Average size (in sectors) of the LUN’s I/O requests.
<code>avgqu-sz</code>	Average number of the LUN’s queued I/O requests.
<code>await</code>	Average time (in milliseconds) that a request waits to complete on the LUN; includes queuing delay and service time.
<code>svctm</code>	Average (LUN) service time (in milliseconds) of I/O requests; does not include any queuing delay.
<code>%util</code>	Percentage of CPU time in which I/O requests are made to the LUN.

Table 1: Black-box, OS-level performance metrics collected for analysis.

In Linux, OS-level performance metrics are made available as text files in the `/proc` pseudo file system. Table 1 describes the specific metrics that we collect. We use `sysstat`’s `sadc` program [15] to periodically gather storage and network performance metrics at a sampling interval of one second, and record them in activity files. For storage resources, `sysstat` provides us with the throughput (`tps`, `rd_sec`, `wr_sec`) and latency (`await`, `svctm`) of the file server’s I/O requests to each of the LUNs the file server is attached to. Since our instrumentation is deployed on file servers, we actually observe the compound effect of disk arrays, controllers, attachments, and the file server on the performance of these I/O requests.

In general we find that `await` is the best single metric for problem diagnosis in parallel file systems as it reflects differences in latency due to both (i) component-level delays (e.g., read errors) and (ii) disparities in request queue length, i.e., differences in workload. Since workload disparities also manifest in changes in throughput, instances in which `await` is anomalous but not `rd_sec` and `wr_sec` indicate a component-level problem.

4.1.1 Continuous Instrumentation

As our test-bench experiments in [17] were of relatively short duration ($\sim 600 \text{ s}$), we were able to spawn instances of `sadc` to record activity files for the duration of our experiments, and perform all analysis once the experiments had finished and the activity files were completely written. For Intrepid, we must continuously instrument and collect data, while also periodically performing offline analysis. To do so, we use a custom daemon, `cycle`, to spawn daily instances of `sadc` shortly after midnight UTC, at which time we are able to collect the previous day’s activity files for analysis.

Although the `cycle` daemon performs a conceptually simple task, we have observed a number of practical issues in deployment that motivated the development of robust time-management features. We elaborate on our

experiences with these issues in § 6. To summarize, the present version of `cycle` implements the following features:

- Records activity files with filenames specified with an ISO 8601-formatted UTC timestamp of `sadc`'s start time.
- Creates new daily activity files at 00:00:05 UTC, which allows up to five seconds of clock backwards-correction without creating a second activity file at 23:59 UTC on the previous day.
- Calls `sadc` to record activity files with a number of records determined by the amount of time remaining before 00:00:05 UTC the next day, as opposed to specifying a fixed number of records. This prevents drifts in file-creation time due to accumulating clock corrections. It also allows for the creation of shorter-duration activity files should a machine be rebooted in the middle of the day.

4.2 Anomaly Detection

The purpose of anomaly detection is to determine which storage LUNs are *instantaneously* reflecting anomalous, non-peer behavior. To do so, we use an improved version of the histogram-based approach described in [17].

Inevitably, any diagnosis algorithm has configurable parameters that are based on the characteristics of the data set for analysis, the pragmatic resource constraints, the specific analytical technique being used, and the desired diagnostic accuracy. In the process of explaining our algorithms below, we also explain the intuition behind the settings of some of these parameters.

Overview. To find the faulty component, we peer-compare storage performance metrics across LUNs to determine those with anomalous behavior. We analyze one metric at a time across all LUNs. For each LUN we first perform a moving average on its metric values. We then generate the Cumulative Distribution Function (CDF) of the smoothed values over a time window of *WinSize* samples. We then compute the distance between CDFs for each pair of LUNs, which represents the degree to which LUNs behave differently. We then flag a LUN as anomalous over a window if more than half of its pairwise CDF distances exceed a predefined threshold. We then shift the window by *WinShift* samples, leaving an overlap of *WinSize* – *WinShift* samples between consecutive windows, and repeat the analysis. We classify a LUN to be faulty if it exhibits anomalous behavior for at least *k* of the past $2k - 1$ windows.

Downsampling. As Intrepid occasionally exhibits a light workload with requests often separated by periods of inactivity, we downsample each storage metric to an interval of 15 s while keeping all other diagnosis parameters the same. This ensures that we incorporate a reason-

able quantity of non-zero metric samples in each comparison window to detect asymmetries. It also serves as a scalability improvement by decreasing analysis time, and decreasing storage and (especially) memory requirements. This is a pragmatic consideration, given that the amount of memory required would be otherwise prohibitively large¹.

Since `sadc` records each of our storage metrics as a rate or time average, proper downsampling requires that we compute the metric's cumulative sum, that we then sample (at a different rate), to generate a new average time series. This ensures any work performed between samples is reflected in the downsampled metric just as it is in the original metric. In contrast, sampling the metric directly would lose any such work, which leads to inaccurate peer-comparison. The result of the downsampling operation is equivalent to running `sadc` with a larger sampling interval.

Moving Average Filter. Sampled storage metrics, particularly for heavy workloads, can contain a large amount of high-frequency (relative to sample rate) noise from which it is difficult to observe subtle, but sustained fault manifestations. Thus, we employ a moving average filter with a 15-sample width to remove this noise. As we do not expect faults to manifest in a periodic manner with a periodicity less than 15 samples, this filter should not unintentionally mask fault manifestations.

CDF Distances. We use cumulative histograms to approximate the CDF of a LUN's smoothed metric values. In generating the histograms we use a modified version of the Freedman-Diaconis rule [12] to select the bin size, $BinSize = 2IQR(x)WinSize^{-1/3}$, and number of bins, $Bins = \lceil Range(x)/BinSize \rceil$ where *x* contains samples across *all* LUNs in the time window. Even though the generated histograms contain samples from a single LUN, we compute *BinSize* using samples from all LUNs to ensure that the resulting histograms have compatible bin parameters and, thus, are comparable. Since each histogram contains only *WinSize* samples, we compute *BinSize* using *WinSize* number of observations. Once histograms are generated for each LUN's values, we compute for each pair of histograms *P* and *Q* the (symmetric) distance: $d(P, Q) = \sum_{i=0}^{Bins} |P(i) - Q(i)|$, a scalar value that represents how different two histograms, and thus LUNs, are from each other.

Windowing and Anomaly Filtering. Looking at our test-bench experiments [17], we found that a *WinSize* of ~60 samples encompassed enough data such that our components were observable as peers, while also

¹We frequently ran out of memory when attempting to analyze the data of a single metric, sampling at 1 s, on machines with 4 GB RAM.

maintaining a reasonable diagnosis latency. We use a *WinShift* of 30 samples between each window to ensure a sufficient window overlap (also 30 samples) so as to provide continuity of behavior from an analysis standpoint. We classify a LUN as faulty if it shows anomalous behavior for 3 out of the past 5 windows ($k = 3$). This filtering process reduces many of the spurious anomalies associated with sporadic asymmetry events where no underlying fault is actually present, but adds to the diagnosis latency. The *WinSize*, *WinShift*, and k values that we use, along with our moving-average filter width, were derived from our test-bench experiments as having providing the best empirical accuracy rates and are similar to the values we published in [17], while also providing for analysis windows that are round to the half-minute. The combined effects of downsampling, windowing, and anomaly filtering result in a diagnosis latency (the time from initial incident to diagnosis) of 22.5 minutes.

4.2.1 Threshold Selection

The CDF distance thresholds used to differentiate faulty from fault-free LUNs are determined through a fault-free training phase that captures the maximum expected deviation in LUN behavior. We use an entire day's worth of data to train thresholds for Intrepid. This is not necessarily the minimum amount of data needed for training, but it is convenient for us to use since our experiment data is grouped by days. We train using the data from the first (manually observed) fault-free day when the system sees reasonable utilization. If possible, we recommend training during stress tests that consist of known workloads, which are typically performed before a new or upgraded storage system goes into production. We can (and do) use the same thresholds in on-going diagnosis, although retraining would be necessary in the event of a system reconfiguration, e.g., if new LUNs are added. Alternatively we could retrain on a periodic (e.g., monthly) basis as a means to tolerate long-term changes to LUN performance. However, in practice, we have not witnessed a significant increase in spurious anomalies during our 15-month study.

To manually verify that a particular day is reasonably fault-free and suitable for training, we generate, for each peer group, plots of superimposed *awaits* for all LUNs within that peer group. We then inspect these plots to ensure that there is no concerning asymmetry among peers, a process that is eased by the fact that most problems manifest as observable loads in normally zero-valued non-primary peer groups. Even if training data is not perfectly fault-free (either due to minor problems that are difficult to observed from *await* plots, or because no such day exists in which faults are completely absent), the influence of faults is only to dampen alarms on the faulty components; non-faulty components remain unaf-

ected. Thus, we recommend that training data should be sufficiently free from observable problems that an operator would feel comfortable operating the cluster indefinitely in its state at the time of training.

4.2.2 Algorithm Refinements

A peer-comparison algorithm requires the use of some measure that captures the similarity and the dissimilarity in the respective behaviors of peer components. A good measure, from a diagnosis viewpoint, is one that captures the differences between a faulty component and its non-faulty peer in a statistically significant way. In our explorations with Intrepid, we have sought to use robust similarity/dissimilarity measures that are improvements over the ones that we used in [17].

The first of these improvements is the method of histogram-bin selection. In [17] we used Sturges' rule [31] to base the number of histogram bins on *WinSize*. Under both faulty and fault-free scenarios (particularly where a LUN exhibits a small asymmetry), Sturges' rule creates histograms where all data is contained in the first and last bins. Thus, the amount of asymmetry of a specific LUN relative to the variance of all LUNs is lost and not represented in the histogram. In contrast, the Freedman–Diaconis rule selects bin size as a function of the interquartile range (IQR), a robust measure of variance uninfluenced by a small number of outliers. Thus, the number of bins in each histogram adapts to ensure an accurate histogram representation of asymmetries that exceeds natural variance.

One notable concern of the Freedman–Diaconis rule is the lack of a limit on the number of bins. Should a metric include outliers that are orders of magnitude larger than the IQR, then, the Freedman–Diaconis rule will generate infeasibly large histograms, which is problematic as the analysis time and memory requirements both scale linearly with the number of bins. While we found this to not typically be an issue with the *await* metric, *wr_sec* outliers would (attempt) to generate histograms with more than 18 million bins. For diagnosis on Intrepid's storage system, we use a bin limit of 1000, which is the 99th, 91st, and 87th percentiles for *await*, *rd_sec*, and *wr_sec* respectively, and results in a worst-case (all generated with 1000 bins) histogram-computation time that is only twice the average.

The second improvement of this algorithm is its use of CDF distances as a similarity/dissimilarity measure, instead of the Probability Density Functions (PDFs) distances as we used in [17]. Specifically, in [17], we used a symmetric version of Kullback–Leibler (KL) divergence [9] to compute distance using histogram approximations of metric PDFs. This comparison works well when two histograms overlap (i.e., many of their data points lie in overlapping bins). However, where two

Date.Hour:	Value	PG:LUN-server			
20110417.00:	7	2:18.61.102	6	2:15.65.74	5 2:11.55.48 5 2:12.48.49
20110417.01:	14	2:15.65.74	9	2:16.51.84	8 1:6.39.8 8 1:10.03.36
20110417.02:	19	2:15.65.74	17	2:16.51.84	15 2:10.50.35 15 2:21.56.121
20110417.03:	25	2:16.51.84	15	2:15.65.74	14 2:21.56.121 13 2:10.50.35
20110417.04:	33	2:16.51.84	20	2:15.65.74	15 2:21.56.121 13 2:10.50.35
20110417.05:	41	2:16.51.84	22	2:15.65.74	13 2:19.53.110 10 1:16.30.87

Figure 5: Example list of persistently anomalous LUNs. Each hour (row) specifies the most persistent anomalies (columns of accumulator value, peer-group, and LUN-server designation), ordered by decreasing accumulator value.

Feature	Test Bench	Intrepid	Rationale
Separating upgraded components	✗	✓	Tolerates weighted I/O on recently added storage capacity; addresses challenge #3.
Fundamental component for analysis	LUNs	LUN-server attachments	Provides views of LUN utilization across redundant components; improves problem localization; addresses challenges #1 and #5.
cycle daemon	✗	✓	Enables continuous instrumentation with <i>sadc</i> ; addresses challenge #4.
Downsampling	✗	1 s \rightarrow 15 s	Tolerates intermittent data, reduces resource requirements; addresses challenge #1.
Histogram bin selection	Sturges' rule	Freedman–Diaconis rule	Provides accurate representation of asymmetries; improves diagnostic accuracy.
Distance metric	KL Divergence (PDF)	Cumulative Distance (CDF)	Accurate distance for non-overlapping histograms; improves diagnostic accuracy.
Persistence Ordering	✗	✓	Highlight components with long-term problems; addresses challenge #6.

Table 2: Improvements to diagnosis approach as compared to previous work [17].

histograms are entirely non-overlapping (i.e., their data points lie entirely in non-overlapping bins in distinct regions of their PDFs), the KL divergence does not include a measure of the distance between non-zero PDF regions. In contrast, the distance between two metric CDFs *does* measure the distance between the non-zero PDF regions, which captures the degree of the LUN's asymmetry.

4.3 Persistence Ordering

While anomaly detection provides us with a reliable account of instantaneously anomalous LUNs, systems of comparable size to Intrepid with thousands of analyzed components, nearly always exhibit one or more anomalies for any given time window, even in the absence of an observable performance degradation.

Motivation. The fact that anomalies “always exist” is a key fact that requires us to alter our focus as we graduate from test-bench experiments to performing problem diagnosis on real systems. In our test-bench work, instantaneous anomalies were rare and either reflected the presence of our injected faults (which we aimed to observe), or the occurrence of false positive (which we aimed to avoid). However, in Intrepid, “spurious” anomalies (even with anomaly filtering) are common enough that we simply cannot raise alarms on each. It is also not possible to completely avoid the alarms through tweaking of analysis parameters (filter width, *WinSize* and *WinShift*, etc.).

Investigating these spurious anomalies, we find that

many are clear instances of transient asymmetries in our raw instrumentation data, due to occasional but regular events where behavior deviates across LUNs. Thus, for Intrepid, we focus our concern on locating system components that demonstrate long-term, or *persistent* anomalies, because they are suggestive of possible impending component failures or problems that might require manual intervention in order to resolve.

Algorithm. To locate persistent anomalies, it is necessary for us to order the list of anomalous LUNs by a measure of their impact on overall performance. To do so, we maintain a positive-value accumulator for every LUN in which we add one (+1) for each window where the LUN is anomalous, and subtract one (−1, and only if the accumulator is > 0) for each window where the LUN is not. We then present to the operator a list of *persistently* anomalous LUNs that are ordered by decreasing accumulator value, i.e., the top-most LUN in the list is that which has the most number of anomalous windows in its recent history. See Figure 5 for an example list.

4.4 Revisiting our Challenges

Table 2 provides a summary of the changes to our approach as we moved from our test-bench environment to performing problem diagnosis in a large-scale storage system. This combination of changes both adequately addresses the challenges of targeting Intrepid's storage system, and also improves the underlying algorithm.

Analysis Step	Runtime	Memory
Extract activity file contents (<code>dump</code>)	1.8 h	< 10 MB
Downsample and tabulate metrics (<code>table</code>)	7.1 h	1.1 GB
Anomaly Detection (<code>diagprep</code>)	49 m	6.1 GB
Persistence Ordering (<code>diagnose</code>)	1.6 s	36 MB
Total	9.7 h	6.1 GB

Table 3: Resources used in analysis of `await`, for the first four peer groups of the 2011-05-09 data set.

4.5 Analysis Resource Requirements

In this section, we discuss the resources (data volume, computation time, and memory) requirements for the analysis of Intrepid’s storage system.

Data Volume. The activity files generated by `sadc` at a sampling interval of 1 s, when compressed with `xz` [8] at preset level `-1`, generate a data volume of approximately 10 MB per file server, per day. The median-size data set (for the day 2011-05-09) has a total (includes all file servers) compressed size of 1.3 GB. In total, we have collected 624 GB in data sets for 474 days.

Runtime. We perform our analysis offline, on a separate cluster consisting of 2.4 GHz dual-core AMD Opteron 1220 machines, each with 4 GB RAM. Table 3 lists our analysis runtime for the `await` metric, when utilizing a single 2.4 GHz Opteron core, for each step of our analysis for the first four peer groups of the median-size data set. Because each data set (consisting of 24 hours of instrumentation data) takes approximately 9.7 h to analyze, we are able to keep up with data input.

We note that the two steps of our analysis that dominate runtime—extracting activity file contents (which is performed on all metrics at once), and downsampling and tabulation of metrics (includes `await` only)—take long due to our sampling at a 1 s interval. We use the 1 s sample rate for archival purposes, as it is the highest sample rate `sadc` supports. However, we could sample at a 15 s rate directly and forgo the downsampling process, which reduces the extraction time in Table 3 by a factor of 15 and tabulation time to 31 m, yielding a total runtime of approximately 1.4h.

Algorithm Scalability. Our CDF distances are generated through the pairwise comparison of histograms is $O(n^2)$ where n is the number of LUNs in each peer group. Because our four peer groups consist of two sets of 768 and 384 LUNs, and our CDF distances are symmetric, we must perform a total of 736,128 histogram comparisons for each analysis window. In practice, we find that our CDF distances are generated quickly, as illustrated by our Anomaly Detection runtime of 49 m for 192 analysis windows (24 hours of data). Thus, we do not see our pairwise algorithm to be an immediate threat to scalability in terms of analysis runtime. We have also proposed [17] an alternative approach to enable $O(n)$

scalability, but found it unnecessary for use in Intrepid.

Memory Utilization. Table 3 also lists the maximum amount of memory used by each step of our analysis. We use the analysis process’ Resident Set Size (RSS) plus any additional used swap memory to determine memory utilization. The most memory-intensive step of our analysis is Anomaly Detection. Our static memory costs come from the need to store the tabulated raw metrics, moving-average-filtered metrics, and a mapping of LUNs to CDF distances, each of which uses 101 MB of memory. Within each analysis window, we must generate histograms for each of the 2,304 LUNs in all four of our peer groups. With a maximum of 1000 bins, all of the histograms occupy at most 8.8 MB of memory. We also generate 736,128 CDF distances, which occupy 2.8 MB per window. However, we must maintain the CDF distances across all 192 analysis windows for a given 24-hour data set, comprising a total of 539 MB. Using R’s [25] default garbage collection parameters, we find that the steady-state memory use while generating CDF distances to be 1.1 GB. The maximum use of 6.1 GB is transient, happening at the very end when our CDF distances are written out to file. With these memory requirements, we are able to analyze two metrics simultaneously on each of our dual-core machines with 4 GB RAM, using swap memory to back the additional 2–4 GB when writing CDF distances to file.

Diagnosis Latency. Our minimum diagnosis latency, that is, the time from the incident of an event to the time of its earliest report as an anomaly is 22.5 minutes. This figure is derived from our (i) performing analysis at a sampling interval of 15 s, (ii) analyzing in time windows shifted by 30 samples, and (iii) requiring that 3 out of the past 5 windows exhibits anomalous behavior before reporting the LUN itself as anomalous:

$$15 \text{ s/samples} \times 30 \text{ samples/window} \times 3 \text{ windows} = 22.5 \text{ m}$$

This latency is an acceptable figure for a few reasons:

- As a tool to diagnose component-level problems when a system is otherwise performing correctly (although, perhaps at suboptimal performance and reduced availability), the system continues to operate usefully during the diagnosis period. Reductions in performance are generally tolerable until a problem can be resolved.
- This latency improves upon current practice in Intrepid’s storage system, e.g., four-hour automated checks of storage controller availability and daily manual checks of controller logs for misbehavior.
- Gabel et al. [13], which targets a similar problem of finding component-level issues before they grow into full-system failures, uses a diagnosis interval of 24 hours, and thus, considers this latency an acceptable figure.

In circumstances where our diagnosis latency would be unacceptably long, lowering the configurable parameters (sample interval, *WinShift*, and Anomaly Filtering's k value) will reduce latency with a potential for increased reports of spurious anomalies, which itself may be an acceptable consequence if there is external indication that a problem exists, for which we may assist in localization. In general, systems that are sensitive to diagnosis latency may benefit from combining our approach with problem-specific ones (e.g., heartbeats, SLAs, threshold limits, and component-specific monitoring) so as to complement each other in problem coverage.

5 Evaluation: Case Study

Having migrated our analysis approach to meeting the challenges of problem diagnosis on a large-scale system, we perform a case study of Intrepid over a 474-day period from April 13th, 2011 through July 31st, 2012. We use the second day (April 14th, 2011) as our only “training day” for threshold selection.

In this study we analyze both “old” and “new” LUN sets, for the first two LUN-server attachment priority groups. This enables us to observe “lost attachment” faults both with zero/missing data from the lost attachment with the primary file server (priority group 1), and with the new, non-peer workload on the attachment with the secondary file server (priority group 2). We note that, although we do not explicitly study priority groups 3–8, we have observed sufficient file server faults to require use of tertiary and subsequent file server attachments.

5.1 Method of Evaluation

After collecting instrumentation data from Intrepid's file servers, we perform the problem diagnosis algorithm described in § 4.2 on the *await* metric, generating a list of the top 100 persistently anomalous LUNs for each hour of the study.

In generating this list, we use a feedback mechanism that approximates the behavior of an operator using this system in real-time. For every period, we consider the top-most persistent anomaly, and if it has sufficient persistence (e.g., an accumulator value in excess of 100, which indicates that the anomaly has been present for at least half a day, but lower values may be accepted given other contextual factors such as multiple LUNs on the same controller exhibiting anomalies simultaneously), then, we investigate that LUN's instrumentation data and storage-controller logs to determine if there is an outstanding performance problem on the LUN, its storage controller, file-server attachments, or attached file servers.

At the time that a problem is remedied (which we determine through instrumentation data and logs, but would be recorded by an operator after performing the

restorative operation), we zero the accumulator for the affected LUN to avoid masking subsequent problems during the anomaly's “wind-down” time (the time during which the algorithm would continually subtract one from the former anomaly's accumulated value until zero is reached). For anomalies that persist for more than a few days before being repaired, we regenerate the persistent-anomaly list with the affected LUNs removed from the list, and check for additional anomalies that indicate a second problem exists concurrently. If a second problem does exist, we repeat this process.

5.2 Observed Incidents

Using our diagnosis approach, we have uncovered a variety of issues that manifested on Intrepid's storage system performance metrics (and that, therefore, we suspect to be performance problems). Our uncovering of these issues was done through our independent analysis of the instrumentation data, with subsequent corroboration of the incident with system logs, operators, and manual inspection of raw metrics. We have grouped these incidents into three categories.

5.2.1 Lost Attachments

We use the *lost attachments* category to describe any problem whereby a file server no longer routes I/O for a particular LUN, i.e., the attachment between that LUN-server pair is “lost”. Of particular concern are lost primary (or highest priority) attachments as it forces clients to reroute I/O through the secondary file server, which then sees a doubling of its workload. Lost attachments of other priorities may still be significant events, but they are not necessarily performance impacting as they are infrequently used for I/O. We observe four general problems that result in lost attachments: (i) failed (or simply unavailable) file servers, (ii) failed storage controllers, (iii) misconfigured components, and (iv) temporary “bad state” problems that usually resolve themselves on reboot.

Failed Events. Table 4 lists the observed down file-server and failed storage-controller events. The *incident time* is the time at which a problem is observed in instrumentation data or controller logs. *Diagnosis latency* is the elapsed time between *incident time* and when we identify the problem using our method of evaluation (see § 5.1). *Recovery latency* is the elapsed time between *incident time* and when our analysis observes the problems to be recovered by Intrepid's operators. *Device* is the component in the system that is physically closest to the origin of the problem, while the incident's observed manifestation is described in *description*. In particular, “missing data” refers to instrumentation data no longer being available for the specified LUN-server attachment due to the disappearance of the LUN's block device on

Incident Time	Diagnosis Latency	Recovery Latency	Device	Description
2011-07-14 00:00	1.0 h	25.8 d	ddn19a	Controller failed on reboot; missing data on 19.00.105.
2011-08-01 19:00	17.0 h	8.9 d	fs16	File server down; observed load on secondary (fs9).
2011-08-15 07:31	29 m	16.5 d	fs24	File server down; observed load on secondary (fs17).
2011-09-05 03:23	8.6 h	18.5 d	ddn20b	Controller manually failed; missing data on 20.00.117.
2011-09-11 03:22	11.6 h	35.6 h	fs25	File server down; observed load on secondary (fs26).
2011-10-03 03:09	12.9 h	42.5 d	ddn11a	Controller failed on reboot; observed load on secondary (fs41).
2011-10-17 16:57	22.1 h	28.0 d	ddn12a, 20a, 21a	Controllers manually failed; observed loads on secondaries (fs53, 115, 125).
2012-06-14 22:26	7.6 h	3.9 d	ddn8a	Controller manually failed; observed load on secondary (fs20).

Table 4: Storage controller failures and down file server lost attachment events.

Incident Time	Diagnosis Latency	Recovery Latency	Device	Description
2011-05-18 00:21	39 m	49.9 d	fs49, 50, 53, 54	Extremely high await (up to 103 s) due to ddn12 resetting all LUNs, results in GPFS timeouts when accessing some LUNs, which remain unavailable until the affected file servers are rebooted; observed “0” await on 12.48.49.
2011-08-08 19:36	8.4 h	21.9 h	ddn19a	Servers unable to access some or all LUNs due to controller misconfiguration (disabled cache coherency); observed “0” await on 19.50.107.
2011-11-14 19:41	7.6 h	3.9 d	ddn14, 18	Cache coherency fails to establish between coupled controllers after reboot, restricting LUN availability to servers; missing data on 14.41.67 and 18.37.103.
2012-03-05 17:50	3.2 h	4.2 d	fs56	GPFS service not available after file server reboot, unknown reason; observed loads on secondary (fs49).
2012-05-10 03:00	9.0 h	4.7 d	ddn16b	LUNs inaccessible from fs84, 88, unknown reason; fixed on controller reboot; missing data on 16.59.84.
2012-06-13 03:00	3.0 h	8.7 d	ddn11a	LUNs inaccessible from fs42, 45, 46, unknown reason; missing data on 11.69.46.

Table 5: Misconfigured component and temporary “bad state” lost attachment events.

that file server, while a “0” value means the block device is still present, but not utilized for I/O.

The lengthy *recovery latency* for each of these *failed* events is due to the fact that all (except for fs25) required hardware replacements to be performed, usually during Intrepid’s biweekly maintenance window, and perhaps even after consultation and troubleshooting of the component with its vendor. At present, Intrepid’s operators discover these problems with `syslog` monitoring (for file servers) and by polling storage-controller status every four hours. Our *diagnosis latency* is high for file-server issues as we depend on the presence of a workload to diagnose traffic to the secondary attachment. Normally these issues would be observed sooner through missing values, except the instrumentation data itself comes from the down file server, and so, is missing in its entirety at the time of the problem (although the missing instrumentation data is a trivial sign that the file server is not in operation). In general, *failed* events, although they can be diagnosed independently, are important for analysis because they are among the longest-duration, numerous-LUN-impacting problems observed in the system.

Misconfiguration and Bad State Events. Table 5 lists the observed misconfiguration and temporary “bad state”

events that result in lost attachments. We explain the two cache-coherency events as follows: Each storage array consists of two coupled storage-controllers, each attached to four different file servers, and both of which are able to provide access to attached disk arrays in the event of one controller’s failure. However, when both controllers are in healthy operation, they may run in either cache-coherent or non-coherent modes. In cache-coherent mode, all LUNs may be accessed by both controllers (and thus, all eight file servers) simultaneously, as they are expected to by the GPFS-cluster configuration. However, should the controllers enter non-coherent mode (due to misconfiguration or a previous controller problem), then they can only access arrays “owned” by the respective controller, restricting four of the eight file servers from accessing some subset of the controllers’ LUNs.

Cascaded Failure. The most interesting example in the “bad state” events is the GPFS timeouts of May 18th, 2011, a cascaded failure that went unnoticed by Intrepid operators for some time. Until the time of the incident, the ddn12 controllers were suffering from multiple, frequent disk issues (e.g., I/O timeouts) when the controller performed 71 “LUN resets”. At this time, the controller delayed responses to incoming I/O requests for up to

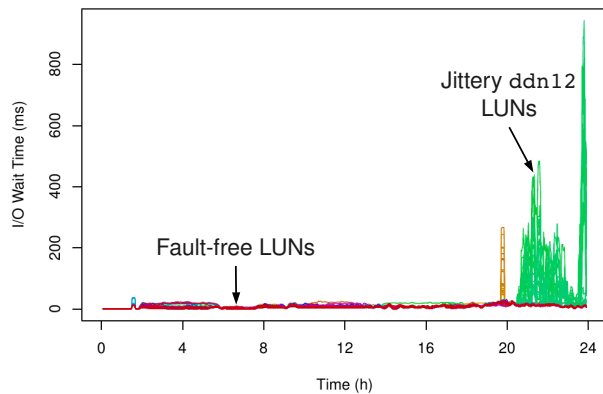


Figure 6: I/O wait time jitter experienced by ddn12 LUNs during the May 17th, 2011 drawer error event.

103 s, causing three of the file servers to timeout their outstanding I/Os and refuse further access to the affected LUNs. Interestingly, while the controller and LUNs remain in operation, the affected file servers continue to abandon access for the duration of 50 days until they are rebooted, at which point the problem is resolved. This particular issue highlights the main benefit of our holistic peer-comparison approach. By having a complete view of the storage system, our diagnosis algorithm is able to locate problems that otherwise escape manual debugging and purpose-specific automated troubleshooting (i.e., scripts written to detect specific problems).

5.2.2 Drawer Errors

A drawer error is an event where a storage controller finds errors, usually I/O and “stuck link” errors on many disks within a single disk drawer. These errors can become very frequent, occurring every few seconds, adding considerable jitter to I/O operations (see Figure 6). Table 6 lists four observed instance of drawer errors, which are fairly similar in their diagnosis characteristics. Drawer errors are visible to operators as a series of many verbose log messages. Operators resolve these errors by forcibly failing every disk in the drawer, rebooting the drawer, then reinserting all the disks into their respective arrays, which are recovered quickly via journal recovery.

5.2.3 Single LUN Events

Single LUN events are instances where a single LUN exhibits considerable I/O wait time (*await*) for as little as a few hours, or as long as many days. Table 7 lists five such events although as many as 40 have been observed to varying extents during our analysis.

These events can vary considerably in their behavior, and Table 7 provides a representative sample. Occasionally, the event will be accompanied by one or more controller-log messages that suggests that one or more

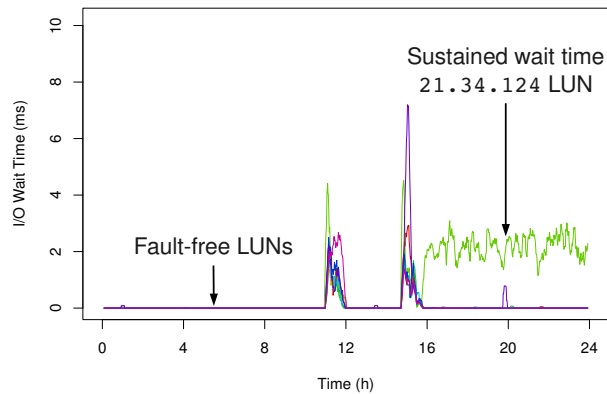


Figure 7: Sustained I/O wait time experienced by 21.34.124 during the June 25th single LUN event.

spindles in the LUN’s disk array is failing, e.g., the June 18th event is accompanied with a message stating that the controller recovered an “8 + 2” parity error. Single LUN events may correspond to single-LUN workloads, and thus, would manifest in one of the throughput metrics (*rd_sec* or *wr_sec*) in addition to *await*. Conversely, the June 25th event in Table 7 manifests in *await* in the *absence* of an observable workload (see Figure 7), perhaps suggesting that there is a load internal to the storage controller or array that causes externally-visible delay. Unfortunately, since storage-controller logs report little on most of our single LUN events, it is difficult to obtain a better understanding of specific causes of these events.

5.3 Alternative Distance Measures

Our use of CDF distances as the distance measure for our peer-comparison algorithm is motivated by its ability to capture the differences in performance metrics between a faulty component and its non-faulty peer. Specifically, CDF distances capture asymmetries in a metric’s value (relative to the metric’s variance across all LUNs), as well as asymmetries in a metric’s shape (i.e., a periodic or increasing/decreasing metric vs. a flat or unchanging metric). The use of CDF distances does require pairwise comparison of histograms, and thus, is $O(n^2)$ where n is the number of LUNs in each peer group. While we have demonstrated that the use of pairwise comparisons is not an immediate threat to scalability (see § 4.5), it is illustrative to compare CDF distances to alternative, computationally-simpler, $O(n)$ distance measures.

The two alternative distance measures we investigate are *median* and *thresh*. For both measures, we use the same Anomaly Detection and Persistence Ordering algorithms as described in § 4.2 and § 4.3, including all windowing, filtering, and their associated parameters. For each time window, instead of generating histograms we use one of our alternative measures to gen-

Incident Time	Diagnosis Latency	Duration	Device	Description
2011-05-17 20:30	3.5 h	3.9 h	ddn12	Jittery await for 60 LUNs due to frequent “G” drawer errors.
2011-06-20 18:30	1.5 h	48.8 h	ddn12	Jittery await for 37 LUNs due to frequent “G” drawer errors.
2011-09-08 02:00	12.0 h	27.0 h	ddn19	Jittery await for 54 LUNs due to frequent “A” drawer errors.
2012-01-16 19:30	3.5 h	24.0 h	ddn16	Jittery await for 11 LUNs due to frequent “D” drawer errors.

Table 6: Drawer error events.

Incident Time	Diagnosis Latency	Duration	Device	Description
2011-06-18 08:00	18.0 h	10.5 d	15.37.78	Sustained above average await; recovered parity errors.
2011-08-18 20:00	26.0 h	79.0 h	19.12.109	Sustained above average await; until workload completes.
2011-09-25 04:00	20.0 h	4.3 d	11.12.45	Sustained above average await; unknown reason.
2012-04-19 12:00	38.0 h	7.2 d	9.04.29	Sustained above average await; unknown reason.
2012-06-25 16:00	8.0 h	6.6 d	21.34.124	Sustained await in absence of workload; unknown reason.

Table 7: Single LUN events.

erate, for each LUN, a scalar distance value from the set of *WinSize* samples. For *median*, we generate a median time-series, *m* by computing for each sample, the median value across all LUNs within a peer group. We then compute each LUN’s scalar distance as the sum of the distances between that LUN’s metric value *x* and the median value for each of the *WinSize* samples: $d(x, m) = \sum_{i=0}^{WinSize} |x(i) - m(i)|$. We then flag a LUN as anomalous over a window if its scalar distance exceeds a predefined threshold, which is selected using the approach described in § 4.2.1.

We follow the same procedure for *thresh*, except that each LUN’s scalar “distance” value is calculated simply as the maximum metric value *x* among the *WinSize* samples: $d(x) = \max x(i) \mid_{i=0}^{WinSize}$. Here, *thresh* is neither truly a measure of *distance*, nor is it being used to perform peer-comparison. Instead we use the *thresh* measure to implement the traditional “metric exceeds alarm-threshold value” within our anomaly detection framework, i.e., an anomalous window using *thresh* indicates that the metric exceeded twice the highest-observed value during the training period for at least one sample.

Performing a meaningful comparison of the *median* and *thresh* measures against CDF distances is challenging with production systems like Intrepid, where our evaluation involves some expert decision making and where we lack ground-truth data. For example, while the events enumerated in Tables 4–7 represent the most significant issues observed in our case study, we know there exists many issues of lesser significance (especially single LUN events) that we have not enumerated. Thus it is not feasible to provide traditional accuracy (true- and false-positive) rates as we have in our test-bench experiments. Instead, we compare the ability of the *median* and *thresh* measures to observe the set of events discovered using CDF distances (listed in Tables 4–7), by following the evaluation procedure described in § 5.1 for the days during which these events occur.

Event Type	CDF	Median	Thresh
Controller failure	5	5	5
File server down	3	2	3
Misconfiguration / bad state	6	6	5
Drawer error	4	3	4
Single LUN	5+	2	1

Table 8: Number of events observed with each distance measure (CDF distances, *median*, and *thresh*).

5.3.1 Comparison of Observations

Table 8 lists the number of we events observe with the alternative distance measures, *median* and *thresh*, as compared to the total events observed with CDF distances. Both *median* and *thresh* measures are able to observe all five failed storage-controller events, as well as most down file-server and misconfiguration/“bad state” events. Each of these events are characterized by missing data on the LUN’s primary attachment, and the appearance of a load on the LUN’s normally-unused secondary attachment. Unlike CDF distances, neither *median* nor *thresh* measures directly account for missing data, however these events are still observed through the presence of secondary-attachment loads. As the non-primary attachments of LUNs are rarely used, these secondary-attachment loads are significant enough to contribute to considerable distance from the (near zero) median and to exceed any value observed during fault-free training. Both measures are also able to observe most drawer errors as these events exhibit considerable peak *await* that exceed both the median value and the maximum-observed value during training.

Controller Misconfiguration. For the August 8th, 2011 controller misconfiguration event, a zero *await* value is observed on the affected LUNs’ primary attachments for the duration of the event, which is observed by the *median* measure. However, this particular event also results in zero *await* on the LUNs’ secondary attachments, which are also affected, pushing the load onto the

LUNs’ tertiary attachments. As we only analyze each LUN’s first two priority groups, the load on the tertiary attachment (and the event itself) goes unobserved. Thus, the *thresh* measure requires analysis of all priority groups to locate “missing” loads that are otherwise directly observed with peer-comparison-based measures.

Single LUN Events. Two single LUN events go unobserved by *median* as their manifestations in increased *await* are not sufficiently higher than their medians to result in persistent anomalies. Four of the events go unobserved by *thresh* as *await* never exceeds its maximum value observed during training, except during the June 25th, 2012 incident on a (normally-unused) secondary attachment where sustained *await* is observed in absence of a workload.

5.3.2 Server Workloads.

The remaining three events that escape observation by *median* (a down file-server, drawer error, and single LUN events) are each due to the same confounding issue. As described in § 3.3.2, shared storage is normally prioritized such that GPFS clients only use the highest-priority available attachment. However, workloads issued by GPFS file servers themselves preferentially make use of their own LUN attachments, regardless of priority, to avoid creating additional LAN traffic. Thus, for server-issued workloads, we observe loads on each (e.g., 48) of the server’s attachments, which span all priority groups, as well as loads on each (e.g., 720) of the primary attachments for LUNs that are not directly attached to those servers. Such workloads, if significant enough, would result in anomalies on each (e.g., 42) of the non-primary attachments.

In practice, Intrepid’s storage system does not run significant, long-running workloads on the file servers, so this complication is usually avoided. The exception is that GPFS itself occasionally issues very low-intensity, long-running (multiple day) workloads from an apparently-random file server. These workloads are of such low intensity (throughput < 10 kB/s, *await* < 1.0 ms, both per LUN) that their *await* values rarely exceed our CDF distance algorithm’s histogram *BinSizes*, and thus, are regarded as noise. However, server-workload *await* values on non-primary attachments do exceed the (zero) median value, and thus, do contribute to *median* anomalies. The result is that the presence of a server workload during an analysis window often exhibits a greater persistence value than actual problems, which confounds our analysis with the *median* measure. Thus, reliable use of the *median* measure requires an additional analysis step to ignore anomalies that appear across all attachments for a particular file server.

Event Type	Median (h)	Thresh (h)
Controller failure	-7, 0, 5, 9, 12	-10, 0, 4, 5, 9
File server down	0, 1	-8, 0, 6
Misconfiguration / bad state	-8, -3, 0, 4, 6, 7	-3, -1, 0, 3, 7
Drawer error	-2, -1, 5	-2, -2, -1, 0
Single LUN	-5, 6	-4

Table 9: Differences in diagnosis latencies for events observed with alternative measures, as compared to CDF.

5.3.3 Comparison of Latencies

Table 9 lists the differences in diagnosis latencies for events observed with the alternative distance measures, *median* and *thresh*, as compared to the diagnosis latencies observed with CDF distances. Negative values indicate that the alternative measure (*median* or *thresh*) observed the event before CDF distances, while positive values indicate that the alternative measure observed the event after. Differences are indicated in integer units as our reporting for the case study is hourly (see Figure 5).

With a mean 1.6 h and median 0.5 h increased latency for *median*, and a mean 0.2 h and median 0 h increased latency for *thresh*, diagnosis latency among all three distance measures are comparable. However, for specific events, latencies can vary as much as twelve hours between measures, suggesting that simultaneous use of multiple measures may be helpful to reduce overall diagnosis latency.

6 Experiences and Insights

In preparing for our case study of Intrepid’s storage system, we made improvements to our diagnosis approach to address the challenges outlined in § 3. However, in the course of our instrumentation and case study, we encountered a variety of pragmatic issues, and we share our experiences and insights with them here.

Clock synchronization. Our diagnosis algorithm requires clocks to be reasonably synchronized across file servers so that we may peer-compare data from the same time intervals. In our test-bench experiments [17], we used NTP to synchronize clocks at the start of our experiments, but disabled the NTP daemon so as to avoid clock adjustments during the experiments themselves. Intrepid’s file servers also run NTP daemons; however, clock adjustments can and do happen during our *sadc* instrumentation. This results in occasional “missing” data samples where the clock adjusts forward, or the occasional “repeat” sample where the clock adjusts backwards. When tabulating data for analysis, we represent missing samples with R’s NA (missing) value, and repeated samples are overwritten with the latest recorded in the activity file. In general, our diagnosis is insensitive to minor clock adjustments and other delays that may result in missing samples, but it is a situation we initially encountered in our *table* script.

Discussion on timestamps. Our activity files are recorded with filenames containing a modified² ISO 8601-formatted [19]³ UTC timestamp that corresponds to the time of the first recorded sample in the file. For example, `fs1-20110509T000005Z.sa.xz` is the activity file collected from file server `fs1`, with the first sample recorded at 00:00:05 UTC on 2011-05-09. In general, we recommend the use of ISO 8601-formatted UTC timestamps for filenames and logging where possible, as they provide the following benefits:

- Human readable (as opposed to Unix time).
- Ensures lexicographical sorting (e.g., of activity files) preserves the chronological order (of records).
- Contains no whitespace, so is easily read as a field by `awk`, R's `read.table`, etc.
- Encodes time zone as a numeric offset; “Z” for UTC.

With regard to time zones, ISO 8601's explicit encoding of them is particularly helpful in avoiding surprises when interpreting timestamps. It is an obvious problem if some components of a system report different time zones than others without expressing their respective zones in timestamps. However, even when all components use the same time zone (as Intrepid uses UTC), offline analysis may use timestamp parsing routines that interpret timestamps without an explicit time-zone designation in the local time zone of the analysis machine (which, in our case, is US Eastern).

A more troubling problem with implicit time zones is that any timestamp recorded during the “repeating” hour of transition from daylight savings time to standard time (e.g., 1 am CDT to 1 am CST) are ambiguous. Although this problem happens only once a year, it causes difficulty in correlating anomalies observed during this hour with event logs from system components that lack time zone designations. Alternatively, when components do encode time zones in timestamps, ISO 8601's use of numeric offsets makes it easy to convert between time zones without needing to consult a time zone database to locate the policies (e.g, daylight savings transition dates) behind time-zone abbreviations.

In summary, ISO 8601 enables easy handling of human readable timestamps without having to work-around edge cases inevitable when performing continuous instrumentation and monitoring of system activity. “Seconds elapsed since epoch” time (e.g., Unix time) works well as a non-human readable alternative as long as the epoch is unambiguous. `sadc` records timestamps in Unix time, and we have had no trouble with them.

²We remove colons to ensure compatibility with file systems that use colons as path separators.

³RFC 3339 is an “Internet profile of the ISO 8601 standard,” that we cite due to its free availability and applicability to computer systems.

Absence of data. One of the surprising outcomes of our case study is that the *absence of*, or “missing data” where it is otherwise expected among its peers, is the primary indication of problem in five (seven if also including “0” data) of the studied events. This result reflects on the effectiveness of peer-comparison approaches for problem diagnosis as they highlight differences in behavior across components. In contrast, approaches that rely on thresholding of raw metric values may not indicate that problems were present in these scenarios.

Separation of instrumentation from analysis. Our diagnosis for Intrepid's storage system consists of simple, online instrumentation, in conjunction with more complex, offline analysis. We have found this separation of online instrumentation and offline analysis to be beneficial in our transition to Intrepid. Our instrumentation, consisting of a well-known daemon (`sadc`), and a small, C-language auditable tool (`cyclops`), have few external dependencies and negligible overhead, both of which are important properties to operators considering deployment on a production system. In contrast, our analysis has significant resource requirements and external dependencies (e.g., the R language runtime and associated libraries), and so is better suited to run on a dedicated machine isolated from the rest of the system. We find that this separation provides an appropriate balance in stability of instrumentation and flexibility in analysis, such that, as we consider “near real-time” diagnosis on Intrepid's storage system, we prefer to maintain the existing design instead of moving to a full-online approach.

7 Future Work

While our persistence-ordering approach works well to identify longer-term problems in Intrepid, there is a class of problems that escapes our current approach. Occasionally, storage controllers will greatly delay I/O processing in response to an internal problem, such as the “LUN resets” observed on `ddn12` in the May 18th, 2011 cascaded failure event. Although we observed this particular incident, in general, order-of-magnitude increases in I/O response times are not highlighted as we ignore the severity of an instantaneous anomaly. Thus, the development of an ordering method that factors in both the severity of instantaneous anomaly, as well as persistence, would be ideal in highlighting both classes of problems.

We also believe we could improve our current problem-diagnosis implementation (see § 5.1) to further increase its utility for systems as large as Intrepid. For instance, problems in storage controllers tend to manifest in a majority of their exported LUNs, and thus, a single problem can be responsible for as many as 50 of the most persistent anomalies. Extending our approach to recognize that these anomalous LUNs are a part of

the same storage controller, and thus, manifest the same problem, would allow us to collapse them into a single anomaly report. This in turn, would make it considerably easier to discover multiple problems that manifest in the same time period. Combining both of these improvements would certainly help us to expand our problem diagnosis and fault coverage.

8 Related Work

Problem Diagnosis in Production Systems. Gabel et al. [13] applies statistical latent fault detection using machine (e.g., performance) counters to a commercial search engine’s indexing and service clusters, and finds that 20% of machine failures are preceded by an incubation period during which the machine deviates in behavior (analogous to our component-level problems) prior to system failure. Draco [18] diagnoses chronics in VoIP operations of a major ISP by, first, heuristically identifying user interactions likely to have failed, and second, identifying groups of properties that best explain the difference between failed and successful interactions. This approach is conceptually similar to ours in using a two-stage process to identify that (i) problems exists, and (ii) localizing them to the most problematic components. Theia [14] is a visualization tool that analyzes application-level logs and generates visual signatures of job performance, and is intended for use by users to locate and problems they experience in a production Hadoop cluster. Theia shares our philosophy of providing a tool to enable users (who act in a similar capacity to our operators) to quickly discover and locate component-level problems within these systems.

HPC Storage-System Characterization. Darshan [6] is a tool for low-overhead, scalable parallel I/O characterization of HPC workloads. Darshan shares our goal of minimal-overhead instrumentation by collecting aggregate statistical and timing information instead of traces in order to minimize runtime delay and data volume, which enables it to scale to leadership-class systems and be used in a “24/7”, always-on manner. Carns et al. [5] combine multiple sources of instrumentation including OS-level storage device metrics, snapshots of file system contents characteristics (file sizes, ages, capacity, etc.), Darshan’s application-level I/O behavior, and aggregate (system-wide) I/O bandwidth to characterize HPC storage system use and behavior. These characterization tools enable a better understanding of HPC application I/O and storage-system utilization, so that both may be optimized to maximize I/O efficiency. Our diagnosis approach is complementary to these efforts, in that it locates sources of acute performance imbalances and problems within the storage system, but assumes that applications are well-behaved and that the normal, balanced

operation is optimal.

Trace-Based Problem Diagnosis. Many previous efforts have focused on path-based [1, 26, 3] and component-based [7, 20] approaches to problem diagnosis in Internet Services. Aguilera et al. [1] treats components in a distributed system as black-boxes, inferring paths by tracing RPC messages and detecting faults by identifying request-flow paths with abnormally long latencies. Pip [26] traces causal request-flows with tagged messages that are checked against programmer-specified expectations. Pip identifies requests and specific lines of code as faulty when they violate these expectations. Magpie [3] uses expert knowledge of event orderings to trace causal request-flows in a distributed system. Magpie then attributes system-resource utilizations (e.g. memory, CPU) to individual requests and clusters them by their resource-usage profiles to detect faulty requests. Pinpoint [7, 20] tags request flows through J2EE web-service systems, and, once a request is known to have failed, identifies the responsible request-processing components.

In HPC environments, Paradyn [22] and TAU [30] are profiling and tracing frameworks used in debugging parallel applications, and IOVIS [23] and Dinh [10] are retrofitted implementations of request-level tracing in PVFS. However, at present, there is limited request-level tracing available in production HPC storage deployments, and thus, we concentrate on a diagnosis approach that utilizes aggregate performance metrics as a readily-available, low-overhead instrumentation source.

Peer-comparison Based Approaches. Ganesha [24] seeks to diagnose performance-related problems in Hadoop by classifying slave nodes, via clustering of performance metrics, into behavioral profiles which are then peer-compared to indict nodes behaving anomalously. While the node-indictment methods are similar, our work peer-compares a limited set of performance metrics directly (without clustering). Bodik et al. [4] use fingerprints as a representation of state to generally diagnose previously-seen datacenter performance crises from SLA violations. Our work avoids using previously-observed faults, and instead relies on fault-free training data to capture expected performance deviations and peer-comparison to determine the presence, specifically, of storage performance problems. Wang et al. [32] analyzes metric distributions to identify RUBiS and Hadoop anomalies in entropy time-series. Our work also avoids the use of raw-metric thresholds by using peer-comparison to determine the degree of asymmetry between storage components, although we do threshold our distance measure to determine the existence of a fault. PeerWatch [16] peer-compares multiple instances of an application running across different virtual ma-

chines, and uses canonical correlation analysis to filter out workload changes and uncorrelated variables to find faults. We also use peer-comparison and bypass workload changes by looking for performance asymmetries, as opposed to analyzing raw metrics, across file servers.

Failures in HPC and Storage Systems. Studies of HPC and storage-system failures motivate our focus on diagnosing problems in storage-system hardware components. A study of failure data collected over nine-years from 22 HPC systems at Los Alamos National Laboratory (LANL) [28] finds that hardware is the largest root cause of failures at 30–60% across the different systems, with software the second-largest contributor at 5–24%, and 20–30% of failures having unknown cause. The large proportion of hardware failures motivates our concentration on hardware-related failures and performance problems. A field-based study of disk-replacement data covering 100,000 disks deployed in HPC and commercial storage systems [29] finds an annual disk replacement rate of 1–4% across the HPC storage systems, and also finds that hard disks are the most commonly replaced components (at 18–49% of the ten most frequently replaced components) in two of three studied storage systems. Given that disks dominate the number of distinct components in the Intrepid storage system, we expect that disk failures and (intermittent) disk performance problems comprise a significant proportion of hardware-related performance problems, and thus, are worthy of specific attention.

9 Conclusion

We presented our experiences of taking our problem diagnosis approach from proof-of-concept on a 12-server test-bench cluster, and making it work on Intrepid’s production GPFS storage system. In doing so, we analyzed 2304 different component metrics across 474 days, and presented a 15-month case study of problems observed in Intrepid’s storage system. We also shared our challenges, solutions, experiences, and insights towards performing continuous instrumentation and analysis. By diagnosing a variety of performance-related storage-system problems, we have shown the value of our approach for diagnosing problems in large-scale storage systems.

Acknowledgements

We thank our shepherd, Adam Oliner, for his comments that helped us to improve this paper. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

- [1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, Oct. 2003.
- [2] Argonne National Laboratory. Blue Gene / P, Dec. 2007. <https://secure.flickr.com/photos/argonne/3323018571/>.
- [3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, Dec. 2004.
- [4] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: Automated classification of performance crises. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*, Paris, France, Apr. 2010.
- [5] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage*, 7(3):8:1–8:26, Oct. 2011.
- [6] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 characterization of petascale I/O workloads. In *Proceedings of the 1st Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS '09)*, New Orleans, LA, Sept. 2009.
- [7] M. Y. Chen, E. Kıcıman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN '02)*, Bethesda, MD, June 2002.
- [8] L. Collin. XZ utils, Apr. 2013. <http://tukaani.org/xz/>.
- [9] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, New York, NY, Aug. 1991.
- [10] T. D. Dinh. Tracing internal behavior in PVFS. Bachelorarbeit, Ruprecht-Karls-Universität Heidelberg, Heidelberg, Germany, Oct. 2009.
- [11] M. Evans. Self-monitoring, analysis and reporting technology (S.M.A.R.T.). SFF Committee Specification SFF-8035i, Apr. 1996.

- [12] D. Freedman and P. Diaconis. On the histogram as a density estimator: L2 theory. *Probability Theory and Related Fields*, 57(4):453–476, Dec. 1981.
- [13] M. Gabel, A. Schuster, R.-G. Bachrach, and N. Bjørner. Latent fault detection in large scale services. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '12)*, Boston, MA, June 2012.
- [14] E. Garduno, S. P. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. Theia: Visual signatures for problem diagnosis in large hadoop clusters. In *Proceedings of the 26th Large Installation System Administration Conference (LISA '12)*, Washington, DC, Nov. 2012.
- [15] S. Godard. SYSSTAT utilities home page, Nov. 2008. <http://pagesperso-orange.fr/sebastien.godard/>.
- [16] H. Kang, H. Chen, and G. Jiang. Peerwatch: a fault detection and diagnosis tool for virtualized consolidation systems. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC '10)*, Washington, DC, June 2010.
- [17] M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan. Black-box problem diagnosis in parallel file systems. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST '10)*, San Jose, CA, Feb. 2010.
- [18] S. P. Kavulya, S. Daniels, K. Joshi, M. Hiltunen, R. Gandhi, and P. Narasimhan. Draco: Statistical diagnosis of chronic problems in large distributed systems. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '12)*, Boston, MA, June 2012.
- [19] G. Klyne and C. Newman. Date and Time on the Internet: Timestamps. RFC 3339 (Proposed Standard), July 2002.
- [20] E. Kıcıman and A. Fox. Detecting application-level failures in component-based Internet services. *IEEE Transactions on Neural Networks*, 16(5):1027–1041, Sept. 2005.
- [21] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, Portland, OR, Nov. 2009.
- [22] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyne parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, Nov. 2005.
- [23] C. Muelder, C. Sigovan, K.-L. Ma, J. Cope, S. Lang, K. Iskra, P. Beckman, and R. Ross. Visual analysis of I/O system behavior for high-end computing. In *Proceedings of the 3rd Workshop on Large-scale System and Application Performance (LSAP '11)*, San Jose, CA, June 2011.
- [24] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Ganesha: Black-box diagnosis of mapreduce systems. In *Proceedings of the 2nd Workshop on Hot Topics in Measurement & Modeling of Computer Systems (HotMetrics '09)*, Seattle, WA, June 2009.
- [25] R Development Core Team. The R project for statistical computing, Apr. 2013. <http://www.r-project.org/>.
- [26] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, , and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, CA, May 2006.
- [27] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02)*, Monterey, CA, Jan. 2002.
- [28] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance-computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '06)*, Philadelphia, PA, June 2006.
- [29] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, San Jose, CA, Feb. 2007.
- [30] S. S. Shende and A. D. Malony. The Tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, May 2006.
- [31] H. A. Sturges. The choice of a class interval. *Journal of the American Statistical Association*, 21(153):65–66, Mar. 1926.
- [32] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan. Online detection of utility cloud anomalies using metric distributions. In *Proceedings of 12th IEEE/IFIP Network Operations and Management Symposium (NOMS '10)*, Osaka, Japan, Apr. 2010.

dsync: Efficient Block-wise Synchronization of Multi-Gigabyte Binary Data

Thomas Knauth
Technische Universität Dresden

Christof Fetzner
Technische Universität Dresden

Abstract

Backing up important data is an essential task for system administrators to protect against all kinds of failures. However, traditional tools like `rsync` exhibit poor performance in the face of today's typical data sizes of hundreds of gigabytes. We address the problem of *efficient*, periodic, multi-gigabyte state synchronization. In contrast to approaches like `rsync` which determine changes after the fact, our approach tracks modifications online. Tracking obviates the need for expensive checksum computations to determine changes. We track modification at the block-level which allows us to implement a very efficient delta-synchronization scheme. The block-level modification tracking is implemented as an extension to a recent (3.2.35) Linux kernel.

With our approach, named `dsync`, we can improve upon existing systems in several key aspects: disk I/O, cache pollution, and CPU utilization. Compared to traditional checksum-based synchronization methods `dsync` decreases synchronization time by up to two orders of magnitude. Benchmarks with synthetic and real-world workloads demonstrate the effectiveness of `dsync`.

1 Introduction

“Everything fails all the time.” is the *modus operandi* when it comes to provisioning critical IT infrastructure. Although the definition of critical is up for debate, redundancy is key to achieving high availability. Redundancy can be added at many levels. For example, at the hardware level, deploying two network cards per server can allow one network card to fail, yet the server will still be reachable. Performance may be degraded due to the failure but the server is still available.

Adding hardware redundancy is just one piece of the availability puzzle. To ensure data availability in the presence of failures, the data must be replicated. However, synchronizing tens, hundreds, or even thousands of

gigabyte of data across the network is expensive. It is expensive in terms of network bandwidth, if a naïve copy-everything approach is used. It is also expensive in terms of CPU cycles, if a checksum-based delta-copy approach is used. Although a delta-copy minimizes network traffic, it relies on a mechanism to identify differences between two versions of the data in question. Determining the differences after the fact is less efficient than recording modifications while they are happening.

One problem with synchronizing large amounts of data, e.g., for backups, is that the backup operation takes on the order of minutes to hours. As data sets continue to grow, consumer drives now hold up to 4 TB, so does the time required to synchronize them. For example, just reading 4 TB stored on a single spinning disk takes more than 6 hours [14]. Copying hundreds of gigabytes over a typical wide area network for remote backups will proceed at a fraction of the previously assumed 170 MB/s. Ideally, the time to synchronize should be independent of the data size; with the size of updated/added data being the main factor influencing synchronization speed.

The key insight is, that between two synchronizations of a data set, most of the data is unmodified. Hence, it is wasteful to copy the entire data set. Even if the data sets are almost identical, the differences have to be determined. This is done, for example, by computing block-wise checksums. Only blocks with mismatching checksums are transferred. Instead of detecting changes after the fact, we propose to track and record them at run time. Online tracking obviates checksum computations, while still only transferring the changed parts of the data. The benefits of online modification recording are plentiful: (1) minimizes network traffic, (2) no CPU cycles spent on checksum computation, (3) minimizes the data read from and written to disk, and (4) minimizes page cache pollution.

We implemented a prototype of our synchronization solution, named `dsync`, on Linux. It consists of a kernel modification and two complimentary userspace

tools. The kernel extension tracks modifications at the block device level. The userspace tools, `dmextract` and `dmmerge`, allow for easy extraction and merging of the modified block level state.

To summarize, in this paper, we make the following contributions:

- Identify the need for better mechanisms to synchronize large, binary data blobs across the network.
- Propose an extension to the Linux kernel to enable efficient synchronization of large, binary data blobs.
- Extend the Linux kernel with block-level tracking code.
- Provide empirical evidence to show the effectiveness of our improved synchronization method.
- We share with the scientific community all code, measurements, and related artifacts. We encourage other researchers to replicate and improve our findings. The results are available at <https://bitbucket.org/tknauth/devicemapper/>.

2 Problem

The task at hand is to periodically synchronize two physically distant data sets and to do so efficiently. The qualifier “periodically” is important because there is little to optimize for a one-off synchronization. With periodic synchronizations, on the other hand, we can potentially exploit the typical case where the majority of the data is unmodified between successive synchronizations.

There exists no domain-specific knowledge about the data being synchronized, i.e., we have to treat it as a binary blob. Using domain-specific knowledge, such as file system meta-data, alternative optimizations are possible. Synchronization tools routinely use a file’s last modified time to check whether to consider it for a possible transfer.

We are primarily interested in large data sizes of multiple giga- to terabytes. The techniques we present are also applicable to smaller sizes, but the problems we solve with our system are more pressing when data sets are large. One example in the cloud computing environment are virtual machine disks. Virtual machine disks change as a result of the customer starting a virtual machine, performing computation, storing the result, and shutting the virtual machine down again. As a result of the users’ actions, the disk’s contents change over time. However, only a fraction of the entire disk is actually modified. It is the cloud provider’s responsibility to store the virtual machine disk in multiple locations, e.g., for fault tolerance. If one data center becomes unavailable, the customer can restart their virtual machine in a backup data

center. For example, a cloud provider may synchronize virtual machine disks once per day between two data centers A and B. If data center A becomes unavailable, data center B has a copy which is at most 24 hours out of date. If customers need more stringent freshness guarantees, the provider may offer alternative backup solutions to the ones considered in this paper.

Copying the entire data is a simple and effective way to achieve synchronization. Yet, it generates a lot of gratuitous network traffic, which is unacceptable. Assuming an unshared 10 Gigabit Ethernet connection, transferring 100 GB takes about 83 seconds (in theory anyway and assuming an ideal throughput of 1.2 GB/s). However, 10 Gigabit Ethernet equipment is still much more expensive than commodity Gigabit Ethernet. While 10 Gigabit may be deployed inside the data center, wide-area networks with 10 Gigabit are even rarer. Also, network links will be shared among multiple participants – be they data streams of the same applications, different users, or even institutions.

The problem of transmitting large volumes of data over constrained long distance links, is exacerbated by continuously growing data sets and disk sizes. Offsite backups are important to provide disaster recovery and business continuity in case of site failures.

Instead of indiscriminately copying everything, we need to identify the changed parts. Only the changed parts must actually be transmitted over the network. Tools, such as `rsync`, follow this approach. The idea is to compute one checksum for each block of data at the source and destination. Only if there is a checksum mismatch for a block, is the block transferred. While this works well for small data sizes, the checksum computation is expensive if data sizes reach into the gigabyte range.

As pointed out earlier, reading multiple gigabytes from disks takes on the order of minutes. Disk I/O operations and bandwidth are occupied by the synchronization process and unavailable to production workloads. Second, checksum computation is CPU-intensive. For the duration of the synchronization, one entire CPU is dedicated to computing checksums, and unavailable to the production workload. Third, reading all that data from disk interferes with the system’s page cache. The working set of running processes is evicted from memory, only to make place for data which is used exactly *once*. Applications can give hints to the operating system to optimize the caching behavior [3]. However, this is not a perfect solution either, as the OS is free to ignore the advice if it cannot adhere to it. In addition, the application developer must be aware of the problem to incorporate hints into the program.

All this is necessary because there currently is no way of identifying changed blocks without comparing their

checksums. Our proposed solution, which tracks block modifications as they happen, extends the Linux kernel to do just that. The implementation details and design considerations form the next section.

3 Implementation

A block device is a well known abstraction in the Linux kernel. It provides random-access semantics to a linear array of blocks. A block typically has the same size as a page, e.g., 4 KiB (2^{12} bytes). The actual media underlying the block device may consist of even smaller units called sectors. However, sectors are not addressable by themselves. Sectors are typically 512 byte in size. Hence, 8 sectors make up a block. Block devices are generally partitioned and formatted with a file system. For more elaborate use cases, the Linux device mapper offers more flexibility to set up block devices.

3.1 Device mapper

The Linux device mapper is a powerful tool. The device mapper, for example, allows multiple individual block devices to be aggregated into a single logical device. In device mapper terminology, this is called a linear mapping. In fact, logical devices can be constructed from arbitrary contiguous regions of existing block devices. Besides simple aggregation, the device mapper also supports RAID configurations 0 (striping, no parity), 1 (mirroring), 5 (striping with distributed parity), and 10 (mirroring and striping). Another feature, which superficially looks like it solves our problem at hand, is snapshots. Block devices can be frozen in time. All modifications to the original device are re-directed to a special copy-on-write (COW) device. Snapshots leave the original data untouched. This allows, for example, to create consistent backups of a block device while still being able to service write requests for the same device. If desired, the external modifications can later be merged into the original block device. By applying the external modifications to a second (potentially remote) copy of the original device, this would solve our problem with zero implementation effort.

However, the solution lacks in two aspects. First, additional storage is required to temporarily buffer all modifications. The additional storage grows linearly with the number of modified blocks. If, because of bad planning, the copy-on-write device is too small to hold all modifications, the writes will be lost. This is unnecessary to achieve what we are aiming for. Second, because modifications are stored out-of-place, they must also be merged into the original data at the source of the actual copy; in addition to the destination. Due to these limitations we

consider device mapper snapshots as an inappropriate solution to our problem.

Because of the way the device mapper handles and interfaces with block devices, our block-level tracking solution is built as an extension to it. The next section describes how we integrated the tracking functionality into the device mapper,

3.2 A Device Mapper Target

The device mapper's functionality is split into separate targets. Various targets implementing, for example, RAID level 0, 1, and 5, already exist in the Linux kernel. Each target implements a predefined interface laid out in the `target_type`¹ structure. The `target_type` structure is simply a collection of function pointers. The target-independent part of the device mapper calls the target-dependant code through one of the pointers. The most important functions are the constructor (`ctr`), destructor (`dtr`), and mapping (`map`) functions. The constructor is called whenever a device of a particular target type is created. Conversely, the destructor cleans up when a device is dismantled. The userspace program to perform device mapper actions is called `dmsetup`. Through a series of `ioctl()` calls, information relevant to setup, tear down, and device management is exchanged between user and kernel space. For example,

```
# echo 0 1048576 linear /dev/original 0 | \
  dmsetup create mydev
```

creates a new device called `mydev`. Access to the sectors 0 through 1048576 of the `mydev` device are mapped to the same sectors of the underlying device `/dev/original`. The previously mentioned function, `map`, is invoked for every access to the linearly mapped device. It applies the offset specified in the mapping. The offset in our example is 0, effectively turning the mapping into an identity function.

The device mapper has convenient access to all the information we need to track block modifications. Every access to a mapped device passes through the `map` function. We adapt the `map` function of the linear mapping mode for our purposes.

3.3 Architecture

Figure 1 shows a conceptual view of the layered architecture. In this example we assume that the tracked block device forms the backing store of a virtual machine (VM). The lowest layer is the physical block device, for example, a hard disk. The device mapper can be used

¹<http://lxr.linux.no/linux+v3.6.2/include/linux/device-mapper.h#L130>

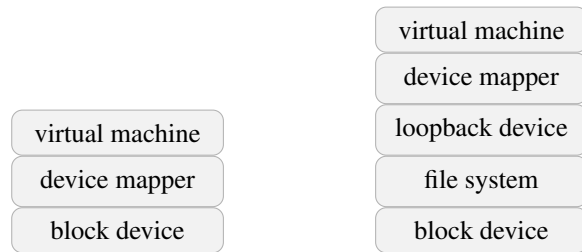


Figure 1: Two configurations where the tracked block device is used by a virtual machine (VM). If the VM used a file of the host system as its backing store, the loopback device turns this file into a block device (right).

to create a tracked device directly on top of the physical block device (Figure 1, left). The tracked block device replaces the physical device as the VM’s backing store.

Often, the backing store of a virtual machine is a file in the host’s filesystem. In these cases, a loopback device is used to convert the file into a block device. Instead of tracking modifications to a physical device, we track modifications to the loopback device (Figure 1, right). The tracked device again functions as the VM’s backing store. The tracking functionality is entirely implemented in the host system kernel, i.e., the guests are unaware of the tracking functionality. The guest OS does not need to be modified, and the tracking works with all guest operating systems.

3.4 Data structure

Storing the modification status for a block requires exactly one bit: a set bit denotes modified blocks, unmodified blocks are represented by an unset bit. The status bits of all blocks form a straightforward bit vector. The bit vector is indexed by the block number. Given the size of today’s hard disks and the option to attach multiple disks to a single machine, the bit vector may occupy multiple megabytes of memory. With 4 KiB blocks, for example, a bit vector of 128 MiB is required to track the per-block modifications of a 4 TiB disk. An overview of the relationship between disk and bit vector size is provided in Table 1.

The total size of the data structure is not the only concern when allocating memory inside the kernel; the size of a single allocation is also constrained. The kernel offers three different mechanisms to allocate memory: (1) `kmalloc()`, (2) `__get_free_pages()`, and (3) `vmalloc()`. However, only `vmalloc()` allows us to reliably allocate multiple megabytes of memory with a single invocation. The various ways of allocating Linux kernel memory are detailed in “Linux Device Drivers” [7].

Total memory consumption of the tracking data structures may still be a concern: even commodity (consumer) machines commonly provide up to 5 SATA ports for attaching disks. Hard disk sizes of 4 TB are standard these days too. To put this in context, the block-wise dirty status for a 10 TiB setup requires 320 MiB of memory. We see two immediate ways to reduce the memory overhead:

1. Increase the minimum unit size from a single block to 2, 4, or even more blocks.
2. Replace the bit vector by a different data structure, e.g., a bloom filter.

A bloom filter could be configured to work with a fraction of the bit vector’s size. The trade-off is potential false positives and a higher (though constant) computational overhead when querying/changing the dirty status. We leave the evaluation of tradeoffs introduced by bloom filters for future work.

Our prototype currently does not persist the modification status across reboots. Also, the in-memory state is lost, if the server suddenly loses power. One possible solution is to persist the state as part of the server’s regular shutdown routine. During startup, the system initializes the tracking bit vector with the state written at shutdown. If the initialization state is corrupt or not existing, each block is marked “dirty” to force a full synchronization.

3.5 User-space interface

The kernel extensions export the relevant information to user space. For each device registered with our customized device mapper, there is a corresponding file in `/proc`, e.g., `/proc/mydev`. Reading the file gives a human-readable list of block numbers which have been written. Writing to the file resets the information, i.e., it clears the underlying bit vector. The `/proc` file system integration uses the `seq_file` interface [15].

Extracting the modified blocks from a block device is aided by a command line tool called `dmextract`. The `dmextract` tool takes as its only parameter the name of the device on which to operate, e.g., `# dmextract mydevice`. By convention, the block numbers for `mydevice` are read from `/proc/mydevice` and the block device is found at `/dev/mapper/mydevice`. The tool outputs, via standard out, a sequence of *(blocknumber,data)* pairs. Output can be redirected to a file, for later access, or directly streamed over the network to the backup location. The complementing tool for block integration, `dmmerge`, reads a stream of information as produced by `dmextract` from standard input. A single parameter points to the block device into which the changed blocks shall be integrated.

Disk size	Disk size (bytes)	Bit vector size (bits)	Bit vector size (bytes)	Bit vector size (pages)	Bit vector size
4 KiB	2^{12}	2^0	2^0	2^0	1 bit
128 MiB	2^{27}	2^{15}	2^{12}	2^0	4 KiB
1 GiB	2^{30}	2^{18}	2^{15}	2^3	64 KiB
512 GiB	2^{39}	2^{27}	2^{24}	2^{12}	16 MiB
1 TiB	2^{40}	2^{28}	2^{25}	2^{13}	32 MiB
4 TiB	2^{42}	2^{30}	2^{27}	2^{15}	128 MiB

Table 1: Relationship between data size and bit vector size. The accounting granularity is 4 KiB, i.e., a single block or page.

Following the Unix philosophy of chaining together multiple programs which each serve a single purpose well, a command line to perform a remote backup may look like the following:

```
# dmextract mydev | \
ssh remotehost dmmerge /dev/mapper/mydev
```

This extracts the modifications from mydev on the local host, copies the information over a secure channel to a remote host, and merges the information on the remote host into an identically named device.

4 Evaluation

The evaluation concentrates on the question of how much the synchronization time decreases by knowing the modified blocks in advance. We compare dsync with four other synchronization methods: (a) copy, (b) rsync, (c) blockmd5sync, (d) ZFS send/receive. Blockmd5sync is our custom implementation of a lightweight rsync. The following sections cover each tool/method in more detail.

4.1 Synchronization tools

4.1.1 scp/nc

scp, short for secure copy, copies entire files or directories over the network. The byte stream is encrypted, hence *secure* copy, putting additional load on the end-point CPUs of the transfer. Compression is optional and disabled for our evaluation. The maximum throughput over a single encrypted stream we achieved with our benchmark systems was 55 MB/s using the (default) aes128-ctr cipher. This is half of the maximum throughput of a 1 gigabit Ethernet adapter. The achievable network throughput for our evaluation is CPU-bound by the single threaded SSH implementation. With a patched version of ssh² encryption can be parallelized for some

ciphers, e.g., aes128-ctr. The application level throughput of this parallelized version varies between 70 to 90 MB/s. Switching to a different cipher, for example, aes128-cbc, gives an average throughput of 100 MB/s.

To transfer data unencrypted, nc, short for netcat, can be used as an alternative to ssh. Instead of netcat, the patched version of ssh also supports unencrypted data transfers. Encryption is dropped after the initial secure handshake, giving us a clearer picture of the CPU requirements for each workload. The throughput for an unencrypted ssh transfer was 100 MB/s on our benchmark systems. We note, however, that whilst useful for evaluation purposes, disabling encryption in a production environment is unlikely to be acceptable and has other practical disadvantages, for example, encryption also helps to detect (non-malicious) in-flight data corruption.

4.1.2 rsync

rsync is used to synchronize two directory trees. The source and destination can be remote in which case data is transferred over the network. Network transfers are encrypted by default because rsync utilizes secure shell (ssh) access to establish a connection between the source and destination. If encryption is undesirable, the secure shell can be replaced by its unencrypted sibling, rsh, although we again note that this is unlikely to be acceptable for production usage. Instead of rsh, we configured rsync to use the drop-in ssh replacement which supports unencrypted transfers. rsync is smarter than scp because it employs several heuristics to minimize the transferred data. For example, two files are assumed unchanged if their modification time stamp and size match. For potentially updated files, rsync computes block-wise checksums at the source and destination. In addition to block-wise checksums, the sender computes rolling checksums. This allows rsync to efficiently handle shifted content, e.g., a line deleted from a configuration file. However, for binary files this creates a huge computational overhead. Only if the checksums for a block are different is that block transferred to the

²<http://www.psc.edu/index.php/hpn-ssh>

destination. For an in-depth description of the algorithm please refer to the work of Tridgell and Mackerras [16]. While `rsync` minimizes the amount of data sent over the network, computing checksums for large files poses a high CPU overhead. Also note, that the checksum computation takes place at both the source *and* destination, although only the source computes rolling checksums.

4.1.3 Blockwise checksums (blockmd5sync)

`rsync`'s performance is limited by its use of rolling checksums at the sender. If we discard the requirement to detect shifted content, a much simpler and faster approach to checksum-based synchronization becomes feasible. We compute checksums only for non-overlapping 4KiB blocks at the sender and receiver. If the checksums for block B_i do not match, this block is transferred. For an input size of N bytes, $\lceil N/B \rceil$ checksums are computed at the source and target, where B is the block size, e.g., 4 kilobytes. The functionality is implemented as a mix of Python and bash scripts, interleaving the checksum computation with the transmission of updated blocks. We do not claim our implementation is the most efficient, but the performance advantages over `rsync` will become apparent in the evaluation section.

4.1.4 ZFS

The file system ZFS was originally developed by Sun for their Solaris operating system. It combines many advanced features, such as logical volume management, which are commonly handled by different tools in a traditional Linux environment. Among these advanced features is snapshot support; along with extracting the difference between two snapshots. We include ZFS in our comparison because it offers the same functionality as `dsync` albeit implemented at a different abstraction level. Working at the file system layer allows access to information unavailable at the block level. For example, updates to paths for temporary data, such as `/tmp`, may be ignored during synchronization. On the other hand, advanced file systems, e.g., like ZFS, may not be available on the target platform and `dsync` may be a viable alternative. As ZFS relies on a copy-on-write mechanism to track changes between snapshots, the resulting disk space overhead must also be considered.

Because of its appealing features, ZFS has been ported to systems other than Solaris ([1], [9]). We use version 0.6.1 of the ZFS port available from <http://zfsonlinux.org> packaged for Ubuntu. It supports the necessary *send* and *receive* operations to extract and merge snapshot deltas, respectively. While ZFS is available on platforms other than Solaris, the port's maturity and reliability may discourage administrators from

adopting it. We can only add anecdotal evidence to this, by reporting one failed benchmark run due to issues within the zfs kernel module.

4.1.5 dsync

Our synchronization tool, `dsync`, differs from `rsync` in two main aspects:

- (a) `dsync` is file-system agnostic because it operates on the block-level. While being file-system agnostic makes `dsync` more versatile, exactly because it requires no file-system specific knowledge, it also constrains the operation of `dsync` at the same time. All the file-system level meta-data, e.g., modification time stamps, which are available to tools like, e.g., `rsync`, are unavailable to `dsync`. `dsync` implicitly assumes that the synchronization target is older than the source.
- (b) Instead of computing block-level checksums at the time of synchronization, `dsync` tracks the per-block modification status at runtime. This obviates the need for checksum calculation between two subsequent synchronizations.

In addition to the kernel extensions, we implemented two userspace programs: One to extract modified blocks based on the tracked information, called `dmextract`. Extracting modified blocks is done at the synchronization source. The equivalent tool, which is run at the synchronization target, is called `dmmerge`. `dmmerge` reads a stream consisting of block numbers interleaved with block data. The stream is merged with the target block device. The actual network transfer is handled either by `ssh`, if encryption is required, or `nc`, if encryption is unnecessary.

4.2 Setup

Our benchmark setup consisted of two machines: one sender and one receiver. Each machine was equipped with a 6-core AMD Phenom II processor, a 2 TB spinning disk (Samsung HD204UI) as well as a 128 GB SSD (Intel SSDSC2CT12). The spinning disk had a 300 GB "benchmark" partition at an outer zone for maximum sequential performance. Except for the ZFS experiments, we formatted the benchmark partition with an `ext3` file system. All benchmarks started and ended with a cold buffer cache. We flushed the buffer cache before each run and ensured that all cached writes are flushed to disk before declaring the run finished. The machines had a Gigabit Ethernet card which was connected to a switch. We ran a recent version of Ubuntu (12.04) with a 3.2 kernel. Unless otherwise noted, each data point is the mean

of three runs. The synchronized data set consisted of a single file of the appropriate size, e.g., 16 GiB, filled with random data. If a tool interfaced with a block device, we created a loopback device on top of the single file.

4.3 Benchmarks

We used two types of benchmarks, synthetic and realistic, to evaluate `dsync`'s performance and compare it with its competitors. While the synthetic benchmarks allow us to investigate worst case behavior, the realistic benchmarks show expected performance for more typical scenarios.

4.3.1 Random modifications

In our synthetic benchmark, we randomly modified varying percentages of data blocks. Each block had the same probability to be modified. Random modification is a worst case scenario because there is little spatial locality. Real world applications, on the other hand, usually exhibit spatial locality with respect to the data they read and write. Random read/write accesses decrease the effectiveness of data prefetching and write coalescing. Because conventional spinning hard disks have a tight limit on the number of input/output operations per second (IOPS), random update patterns are ill suited for them.

4.3.2 RUBiS

A second benchmark measures the time to synchronize virtual machine images. The tracked VM ran the RUBiS [5] server components, while another machine ran the client emulator. RUBiS is a web application modeled after `eBay.com`. The web application, written in PHP, consists of a web server, and a data base backend. Users put items up for sale, bid for existing items or just browse the catalog. Modifications to the virtual machine image resulted, for example, from updates to the RUBiS data base.

A single run consisted of booting the instance, subjecting it to 15 minutes of simulated client traffic, and shutting the instance down. During the entire run, we recorded all block level updates to the virtual machine image. The modified block numbers were the input to the second stage of the experiment. The second stage used the recorded block modification pattern while measuring the synchronization time. Splitting the experiment into two phases allows us to perform and repeat them independently.

4.3.3 Microsoft Research Traces

Narayanan et al. [11] collected and published block level

traces for a variety of servers and services at Microsoft Research³. The traces capture the block level operations of, among others, print, login, and file servers. Out of the available traces we randomly picked the print server trace. Because the print server's non-system volume was several hundred gigabytes in size, we split the volume into non-overlapping, 32 GiB-sized ranges. Each operation in the original trace was assigned to exactly one range, depending on the operation's offset. Further analysis showed that the first range, i.e., the first 32 GiB of the original volume, had the highest number of write operations, close to 1.1 million; more than double of the second "busiest" range.

In addition to splitting the original trace along the space axis, we also split it along the time axis. The trace covers over a period of seven days, which we split into 24 hour periods.

To summarize: for our analysis we use the block modification pattern for the first 32 GiB of the print server's data volume. The seven day trace is further divided into seven 24 hour periods. The relative number of modified blocks is between 1% and 2% percent for each period.

4.4 Results

4.4.1 Random modifications

We start the evaluation by looking at how the data set size affects the synchronization time. The size varies between 1 and 32 GiB for which we randomly modified 10% of the blocks. The first set of results is shown in Figure 2. First of all, we observe that the synchronization time increases linearly with the data set size; irrespective of the synchronization method. Second, `rsync` takes longest to synchronize, followed by `blockmd5sync` on HDD and copy on SSD. Copy, ZFS, and `dsync` are fastest on HDD and show similar performance. On SSD, `dsync` and `blockmd5sync` are fastest, with ZFS being faster than copy, but not as fast as `dsync` and `blockmd5sync`. With larger data sizes, the performance difference is more markedly: for example, at 32 GiB `dsync`, copy, and ZFS perform almost identically (on HDD), while `rsync` takes almost five times as long (420 vs. 2000 seconds). To our surprise, copying the entire state is sometimes as fast as or even slightly faster than extracting and merging the differences. Again at 32 GiB, for example, copy takes about 400 seconds, compared with 400 seconds for `dsync` and 420 seconds for ZFS.

We concluded that the random I/O operations were inhibiting `dsync` to really shine. Hence, we performed a second set of benchmarks where we used SSDs instead of HDDs. The results are shown in Figure 3. While

³available at <ftp://ftp.research.microsoft.com/pub/austind/MSRC-io-traces/>

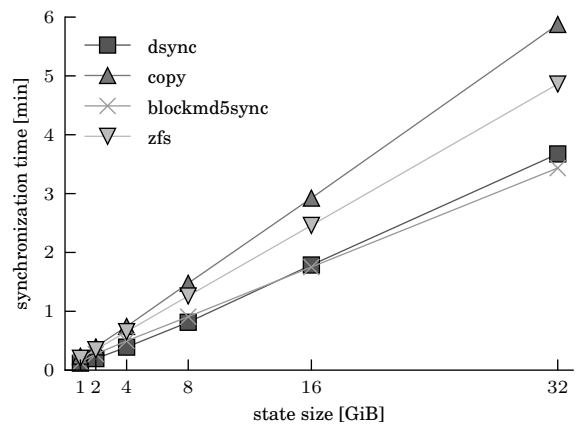
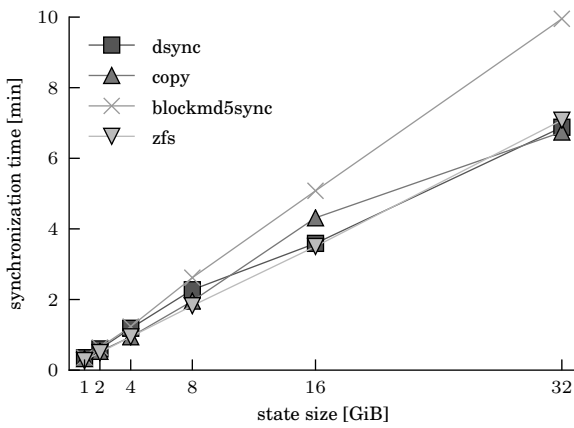
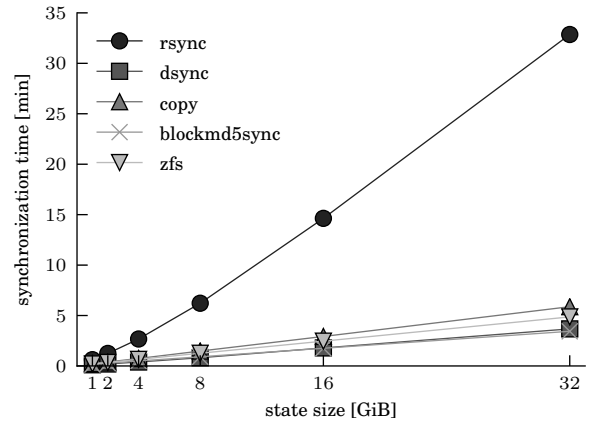
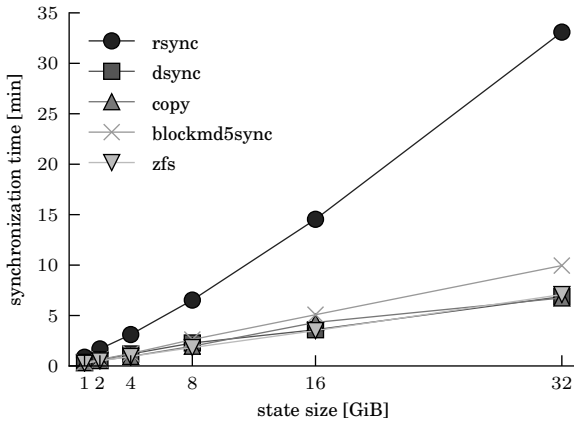


Figure 2: Synchronization time for five different synchronization techniques. Lower is better. Data on the source and target was stored on HDD.

Figure 3: Synchronization time for five different synchronization techniques. Lower is better. Data on the source and target was stored on SSD.

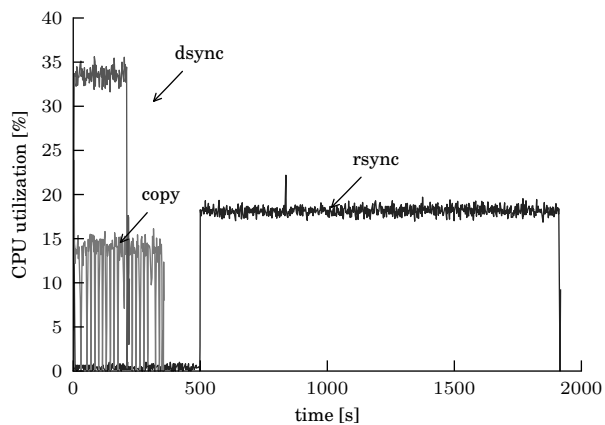


Figure 4: CPU utilization for a sample run of three synchronization tools. 100% means all cores are busy.

the increased random I/O performance of SSDs does not matter for `rsync`, its synchronization time is identical to the HDD benchmark, SSDs enable all other methods to finish faster. `dsync`'s time to synchronize 32 GiB drops from 400 s on HDD to only 220 s on SSD.

Intrigued by the trade-off between hard disk and solid state drives, we measured the read and write rate of our drives outside the context of `dsync`. When extracting or merging modified blocks they are processed in increasing order by their block number. We noticed that the read/write rate increased by up to 10x when processing a *sorted* randomly generated sequence of block numbers compared to the same *unsorted* sequence. For a random but sorted sequence of blocks our HDD achieves a read rate of 12 MB/s and a write rate of 7 MB/s. The SSD reads data twice as fast at 25 MB/s and writes data more than 15x as fast at 118 MB/s. This explains why, if HDDs are involved, `copy` finishes faster than `dsync` although `copy`'s transfer volume is 9x that of `dsync`: sequentially going through the data on HDD is much faster than selectively reading and writing only changed blocks.

To better highlight the differences between the methods, we also present CPU and network traffic traces for three of the five methods. Figure 4 shows the CPU utilization while Figure 5 shows the outgoing network traffic at the sender. The trace was collected at the sender while synchronizing 32 GiB from/to SSD. The CPU utilization includes the time spent in kernel and user space, as well as waiting for I/O. We observe that `rsync` is CPU-bound by its single-threaded rolling checksum computation. Up to $t = 500$ the `rsync` sender process is idle, while one core on the receiver-side computes check-

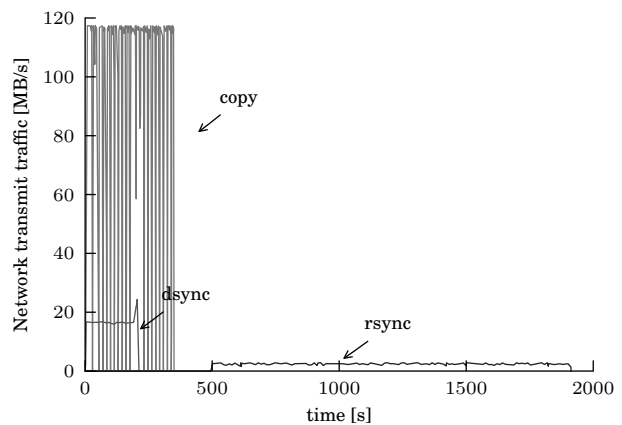


Figure 5: Network transmit traffic on the sender side measured for the entire system. `rsync` and `dsync` transmit about the same amount of data in total, although the effective throughput of `rsync` is much lower.

sums (not visible in the graph). During `rsync`'s second phase, one core, on our 6-core benchmark machine, is busy computing and comparing checksums for the remaining 1400 s (23 min). The network traffic during that time is minimal at less than 5 MB/s. `Copy`'s execution profile taxes the CPU much less: utilization oscillates between 0% and 15%. On the other hand, it can be visually determined that `copy` generates much more traffic volume than either `rsync` or `dsync`. `Copy` generates about 90 MB/s of network traffic on average. `dsync`'s execution profile uses double the CPU power of `copy`, but only incurs a fraction of the network traffic. `dsync`'s network throughput is limited by the random read-rate at the sender side.

Even though the SSD's specification promises 22.5 k random 4 KiB reads [2], we are only able to read at a sustained rate of 20 MB/s at the application layer. Adding a loopback device to the configuration, reduces the application layer read throughput by about another 5 MB/s. This explains why `dsync`'s sender transmits at 17 MB/s. In this particular scenario `dsync`'s performance is read-limited. Anything that would help with reading the modified blocks from disk faster, would decrease the synchronization time even further.

Until now we kept the modification ratio fixed at 10%, which seemed like a reasonable change rate. Next we explore the effect of varying the percentage of modified blocks. We fix the data size at 8 GiB and randomly modify 10%, 50%, and 90% percent of the blocks. Figure 6 and 7 show the timings for spinning and solid-state disks. On HDD, interestingly, even though the amount of data sent across the network increases, the net

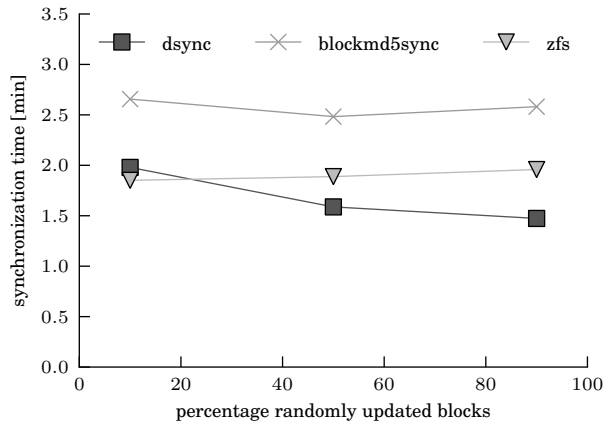


Figure 6: For comparison, rsync synchronizes the same data set 6, 21, and 41 minutes, respectively. Copy took between 1.5 and 2 minutes.

synchronization time stays almost constant for ZFS and blockmd5sync; it even *decreases* for dsync. Conversely, on SSD, synchronization takes longer with a larger number of modified blocks across all shown methods; although only minimally so for ZFS. We believe the increase for dsync and blockmd5sync is due to a higher number of block-level re-writes. Updating a block of flash memory is expensive and often done in units larger than 4 KiB [8]. ZFS is not affected by this phenomenon, as ZFS employs a copy-on-write strategy which turns random into sequential writes.

4.4.2 RUBiS results

We argued earlier, that a purely synthetic workload of random block modifications artificially constraints the performance of dsync. Although we already observed a 5x improvement in total synchronization time over rsync, the gain over copy was less impressive. To highlight the difference in spatial locality between the synthetic and RUBiS benchmark, we plotted the number of consecutive modified blocks for each; prominently illustrated in Figure 8.

We observe that 80% of the modification involve only a single block (36k blocks at $x = 1$ in Figure 8). In comparison, there are no single blocks for the RUBiS benchmark. Every modification involves at least two consecutive blocks (1k blocks at $x = 2$). At the other end of the spectrum, the longest run of consecutively modified blocks is 639 for the RUBiS benchmarks. Randomly updated blocks rarely yield more than 5 consecutively modified blocks. For the RUBiS benchmark, updates of 5 consecutive blocks happen most often: the total number of modified blocks jumps from 2k to 15k moving from 4

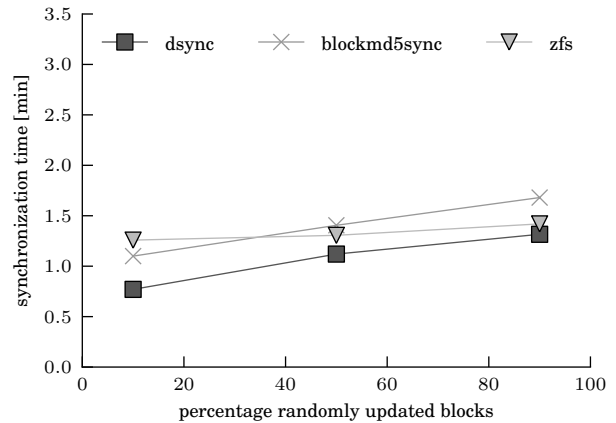


Figure 7: Varying the percentage of modified blocks for an 8 GiB file/device. For comparison, rsync synchronizes the same data set in 5, 21, and 41 minutes, respectively. A plain copy consistently took 1.5 minutes.

to 5 consecutively modified blocks.

Now that we have highlighted the spatial distribution of updates, Figure 9 illustrates the results for our RUBiS workload. We present numbers for the HDD case only because this workload is less constrained by the number of I/O operations per second. The number of modified blocks was never the same between those 20 runs. Instead, the number varies between 659 and 3813 blocks. This can be explained by the randomness inherent in each RUBiS benchmark invocation. The type and frequency of different actions, e.g., buying an item or browsing the catalog, is determined by chance. Actions that modify the data base increase the modified block count.

The synchronization time shows little variation between runs of the same method. Copy transfers the entire 11 GiB of data irrespective of actual modifications. There should, in fact, be no difference based on the number of modifications. rsync's execution time is dominated by checksum calculations. dsync, however, transfers only modified blocks and should show variations. The relationship between modified block count and synchronization time is just not discernible in Figure 9. Alternatively, we calculated the correlation coefficient for dsync which is 0.54. This suggests a positive correlation between the number of modified blocks and synchronization time. The correlation is not perfect because factors other than the raw modified block count affect the synchronization time, e.g., the spatial locality of updates.

The performance in absolute numbers is as follows: rsync, which is slowest, takes around 320 seconds to synchronize the virtual machine disk. The runner up, copy, takes 200 seconds. The clear winner, with an av-

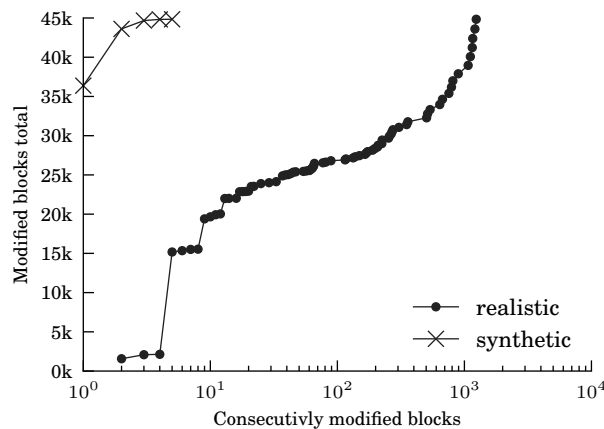


Figure 8: Difference in spatial locality between a synthetic and realistic benchmark run. In both cases 45k blocks are modified. For the synthetic benchmark 80% are isolated individual blocks (36k at $x=1$). The realistic benchmark shows a higher degree of spatial locality, as observed, for example, by the jump from 2.5k ($x=3$) to 15k ($x=4$) blocks.

erage synchronization time of about 3 seconds, is `dsync`. That is a factor 66x improvement over `copy` and more than 100x faster than `rsync`. `dsync` reduces the network traffic to a minimum, like `rsync`, while being 100x faster. Table 2 summarizes the results.

4.4.3 Microsoft Research Traces

In addition to our benchmarks with synchronizing a single virtual machine disk, we used traces from a Microsoft Research (MSR) printer server. The speed with which the different methods synchronize the data is identical across all days of the MSR trace. Because of the homogeneous run times, we only show three days out of the total seven in Figure 10.

`rsync` is slowest, taking more than 900 seconds (15 minutes) to synchronize 32 GiB of binary data. The small number of updated blocks (between 1-2%) decreases the runtime of `rsync` noticeably. Previously, with 10% updated blocks `rsync` took 35 minutes (cf. Figure 2) for the same data size. `Copy` and `blockmd5sync` finish more than twice as fast as `rsync`, but are still considerably slower than either `ZFS` or `dsync`. The relative order of synchronization times does not change when we swap HDDs for SSDs (Figure 11). Absolute synchronization times improve for each synchronization method. `blockmd5sync` sees the largest decrease as its performance is I/O bound on our setup: the SSD offers faster sequential read speeds than the HDD, 230 MB/s vs 130 MB/s.

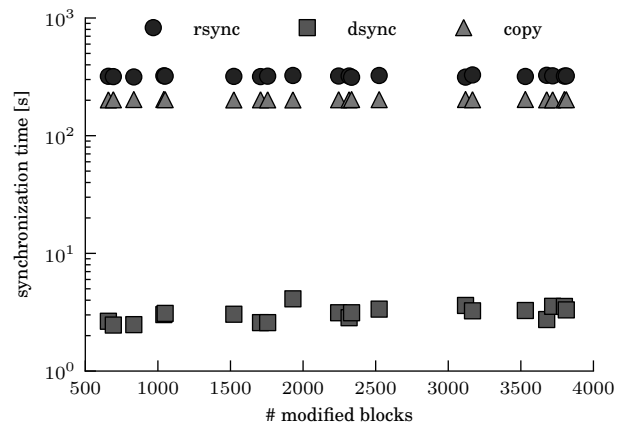


Figure 9: Synchronization time for (a) `copy`, (b) `rsync`, and (c) `dsync`. Block modifications according to the RUBiS workload.

4.5 Discussion

One interesting question to ask is if there exist cases where `dsync` performs worse than `rsync`. For the scenarios investigated in this study `dsync` always outperformed `rsync` by a significant margin. In fact, we believe, that `dsync` will always be faster than `rsync`. Our reasoning is that the operations performed by `rsync` are a superset of `dsync`'s operations. `rsync` must read, transmit, and merge all the updated blocks; as does `dsync`. However, to determine the updated blocks `rsync` must read *every* block and compute its checksum at the source *and* destination. As illustrated and mentioned in the capture for Figure 4, the computational overhead varies with the number of modified blocks. For identical input sizes, the execution time of `rsync` grows with the number of updated blocks.

The speed at which `dsync` synchronizes depends to a large degree on the spatial distribution of the modified blocks. This is most visible in Figures 6. Even though the data volume increases by 5x, going from 10% randomly modified blocks to 50%, the synchronization takes *less* time. For the scenarios evaluated in this paper, a simple `copy` typically (cf. Figure 6, 2) took at least as long as `dsync`. While `dsync` may not be faster than a plain `copy` in all scenarios, it definitely reduces the transmitted data.

Regarding the runtime overhead of maintaining the bitmap, we do not expect this to noticeably affect performance in typical use cases. Setting a bit in memory is orders of magnitude faster than actually writing a block to disk.

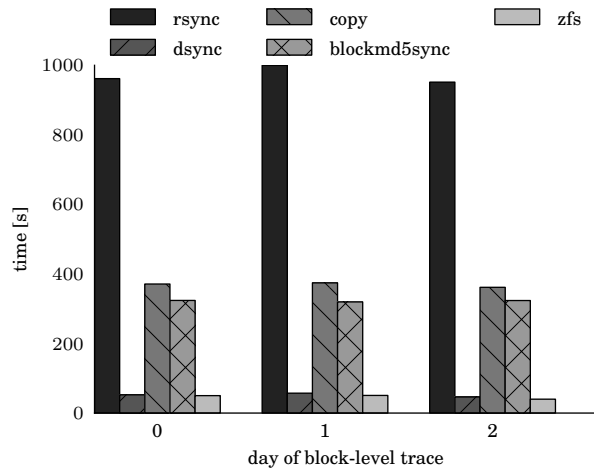


Figure 10: Synchronization times for realistic block-level update patterns on HDDs. Lower is better.

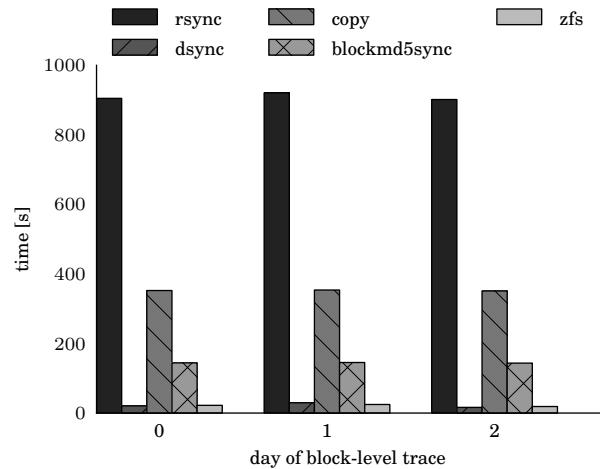


Figure 11: Synchronization times for realistic block-level update patterns on SSDs. Lower is better.

5 Related work

File systems, such as ZFS, and only recently btrfs, also support snapshots and differential backups. In ZFS lingo the operations are called *send* and *receive*. The delta between two snapshots can be extracted and merged again with another snapshot copy, e.g., at a remote backup machine. Only users of ZFS, however, can enjoy those features. For btrfs, there exists a patch to extract differences between snapshot states [6]. This feature is, however, still considered experimental. Besides the file system, support for block tracking can be implemented higher up still in the software stack. VMware ESX, since version 4, is one example which supports block tracking at the application layer. In VMware ESX server the feature is called *changed block tracking*. Implementing support for efficient, differential backups at the block-device level, like dsync does, is more general, because it works regardless of the file system and application running on top.

If updates must be replicated more timely to reduce the inconsistency window, the distributed replicated block device (DRBD) synchronously replicates data at the block level. All writes to the primary block device are mirrored to a second, standby copy. If the primary block device becomes unavailable, the standby copy takes over. In single primary mode, only the primary receives updates which is required by file systems lacking concurrent access semantics. Non-concurrent file systems assume exclusive ownership of the underlying device and single primary mode is the only viable DRBD configuration in this case. However, DRBD also supports dual-primary configurations, where both copies receive updates. A dual-primary setup requires a concurrency-aware file system, such as GFS or OCFS, to maintain consistency. DRBD is part of Linux since kernel version

2.6.33.

There also exists work to improve the efficiency of synchronization tools. For example, Rasch and Burns [13] proposed for *rsync* to perform in-place updates. While their intention was to improve *rsync* performance on resource-constraint mobile devices, it also helps with large data sets on regular hardware. Instead of creating an out-of-place copy and atomically swapping this into place at the end of the transfer, the patch performs in-place updates. Since their original patch, in-place updates have been integrated into regular *rsync*.

A more recent proposal tackles the problem of *page cache pollution* [3]. During the backup process many files and related meta-data are read. To improve system performance, Linux uses a page cache, which keeps recently accessed files in main memory. By reading large amounts of data, which will likely *not* be accessed again in the near future, the pages cached on behalf of other processes, must be evicted. The above mentioned patch reduces cache pollution to some extent. The operating system is advised, via the *fsync* system call, that pages, accessed as part of the *rsync* invocation, can be evicted immediately from the cache. Flagging pages explicitly for eviction, helps to keep the working sets of other processes in memory.

Effective buffer cache management was previously discussed, for example, by Burnett et al. [4] and Plonka et al. [12]. Burnett et al. [4] reverse engineered the cache replacement algorithm used in the operating system. They used knowledge of the replacement algorithm at the application level, here a web server, to change the order in which concurrent requests are processed. As a result, the average response time decreases and throughput increases. Plonka et al. [12] adapted their net-

tool	sync time [s]	state transferred [MB]
rsync	950	310
copy	385	32768
blockmd5sync	310	310
ZFS	42	310
dsync	38	310

Table 2: Performance summary for realistic benchmark.

work monitoring application to give the operating system hints about which blocks can be evicted from the buffer cache. Because the application has ultimate knowledge about access and usage patterns, the performance with application-level hints to the OS is superior. Both works agree with us on the sometimes adverse effects of the default caching strategy. Though the strategy certainly improves performance in the average case, subjecting the system to extreme workloads, will reveal that sometimes the default is ill suited.

6 Conclusion

We tackled the task of periodically synchronizing large amounts of binary data. The problem is not so much how to do it, but how to do it *efficiently*. Even today, with widespread home broadband connections, network bandwidth is a precious commodity. Using it to transmit gigabytes of redundant information is wasteful. Especially for data sets in the terabyte range the good old “sneakernet” [10] may still be the fastest transmission mode. In the area of cloud computing, large binary data blobs prominently occur in the form of virtual machine disks and images. Backup of virtual machine disks and images, e.g., fault tolerance, is a routine task for any data center operator. Doing so efficiently and with minimal impact on productive workloads is in the operator’s best interest.

We showed how existing tools, exemplified by `rsync`, are ill-suited to synchronize gigabyte-sized binary blobs. The single-threaded checksum computation employed by `rsync` leads to synchronization times of 32 minutes even for moderate data sizes of 32 GB. Instead of calculating checksums when synchronization is requested, we track modifications on line. To do so, we extended the existing device mapper module in the Linux kernel. For each tracked device, the modified block numbers can be read from user-space. Two supplemental tools, called `dmextract` and `dmmerge`, implement the extraction and merging of modified blocks in user-space. We call our system `dsync`.

A mix of synthetic and realistic benchmarks demon-

strates the effectiveness of `dsync`. In a worst case workload, with exclusively random modifications, `dsync` synchronizes 32 GB in less than one quarter of the time that `rsync` takes, i.e., 7 minutes vs 32 minutes. A more realistic workload, which involves the synchronization of virtual machines disks, reduces the synchronization time to less than 1/100th that of `rsync`.

Acknowledgments

The authors would like to thank Carsten Weinhold, and Adam Lackorzynski for their helpful insights into the Linux kernel, as well as Björn Döbel, Stephan Diestelhorst, and Thordis Kombrink for valuable comments on drafts of this paper. In addition, we thank the anonymous LISA reviewers and in particular our shepherd, Mike Ciavarella. This research was funded as part of the ParaDIME project supported by the European Commission under the Seventh Framework Program (FP7) with grant agreement number 318693.

References

- [1] URL <http://zfsonlinux.org>.
- [2] Intel solid-state drive 330 series: Specification. URL <http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-330-specification.pdf>.
- [3] Improving Linux Performance by Preserving Buffer Cache State. URL <http://insights.oetiker.ch/linux/fadvise/>.
- [4] N. Burnett, J. Bent, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Management. In *Proceedings of USENIX Annual Technical Conference*, pages 29–44, 2002.
- [5] OW2 Consortium. RUBiS: Rice University Bidding System. URL <http://rubis.ow2.org>.
- [6] Jonathan Corbet. Btrfs send/receive. URL <http://lwn.net/Articles/506244/>.
- [7] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O’Reilly Media, 3rd edition edition, 2009.
- [8] Michael Cornwell. Anatomy of a Solid-state Drive. *Queue*, 10(10), 2012.
- [9] Pawel Jakub Dawidek. Porting the ZFS file system to the FreeBSD operating system. *AsiaBSDCon*, pages 97–103, 2007.
- [10] Jim Gray, Wyman Chong, Tom Barclay, Alexander S. Szalay, and Jan vandenBerg. TeraScale SneakerNet: Using Inexpensive Disks for Backup, Archiving, and Data Exchange. *CoRR*, cs.NI/0208011, 2002.
- [11] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. *ACM Transactions on Storage*, 4(3), 2008.

- [12] D. Plonka, A. Gupta, and D. Carder. Application Buffer-Cache Management for Performance: Running the World's Largest MRTG. In *Large Installation System Administration Conference*, pages 1–16, 2007.
- [13] D. Rasch and R. Burns. In-Place Rsync: File Synchronization for Mobile and Wireless Devices. In *Proc. of the USENIX Annual Technical Conference*, 2003.
- [14] David Rosenthal. Keeping Bits Safe: How Hard Can It Be? *Queue*, 8(10), 2010. URL <http://queue.acm.org/detail.cfm?id=1866298>.
- [15] P.J. Salzman, M. Burian, and O. Pomerantz. The Linux Kernel Module Programming Guide. 2001. URL <http://www.tldp.org/LDP/lkmpg/2.6/html/>.
- [16] A. Tridgell and P. Mackerras. The rsync Algorithm. Technical Report TR-CS-96-05, Australian National University, 1996.

HotSnap: A Hot Distributed Snapshot System for Virtual Machine Cluster

Lei Cui, Bo Li, Yangyang Zhang, Jianxin Li
Beihang University, Beijing, China
{cuilei, libo, zhangyy, lijx}@act.buaa.edu.cn

Abstract

The management of virtual machine cluster (VMC) is challenging owing to the reliability requirements, such as non-stop service, failure tolerance, etc. Distributed snapshot of VMC is one promising approach to support system reliability, it allows the system administrators of data centers to recover the system from failure, and resume the execution from a intermediate state rather than the initial state. However, due to the heavyweight nature of virtual machine (VM) technology, applications running in the VMC suffer from long downtime and performance degradation during snapshot. Besides, the discrepancy of snapshot completion times among VMs brings the TCP backoff problem, resulting in network interruption between two communicating VMs. This paper proposes HotSnap, a VMC snapshot approach designed to enable taking *hot* distributed snapshot with milliseconds system downtime and TCP backoff duration. At the core of HotSnap is transient snapshot that saves the minimum instantaneous state in a short time, and full snapshot which saves the entire VM state during normal operation. We then design the snapshot protocol to coordinate the individual VM snapshots into the global consistent state of VMC. We have implemented HotSnap on QEMU/KVM, and conduct several experiments to show the effectiveness and efficiency. Compared to the live migration based distributed snapshot technique which brings seconds of system downtime and network interruption, HotSnap only incurs tens of milliseconds.

1 Introduction

With the increasing prevalence of cloud computing and IaaS paradigm, more and more distributed applications and systems are migrating to and running on virtualization platform. In virtualized environments, distributed applications are encapsulated into virtual machines, which are connected into virtual machine cluster (VM-

C) and coordinated to complete the heavy tasks. For example, Amazon EC2 [1] offers load balancing web farm which can dynamically add or remove virtual machine (VM) nodes to maximize resource utilization; CyberGuarder [22] encapsulates security services such as IDS and firewalls into VMs, and deploys them over a virtual network to provide virtual network security service; Emulab [12] leverages VMC to implement on-demand virtual environments for developing and testing networked applications; the parallel applications, such as map-reduce jobs, scientific computing, client-server systems can also run on the virtual machine cluster which provides an isolated, scaled and closed running environment.

Distributed snapshot [13, 27, 19] is a critical technique to improve system reliability for distributed applications and systems. It saves the running state of the applications periodically during the failure-free execution. Upon a failure, the system can resume the computation from a recorded intermediate state rather than the initial state, thereby reducing the amount of lost computation [15]. It provides the system administrators the ability to recover the system from failure owing to hardware errors, software errors or other reasons.

Since the snapshot process is always carried out periodically during normal execution, transparency is a key feature when taking distributed snapshot. In other words, the users or applications should be unaware of the snapshot process, neither the snapshot implementation scheme nor the performance impact. However, the traditional distributed systems either implement snapshot in OS kernel [11], or modify the MPI library to support snapshot function [17, 24]. Besides, many systems even leave the job to developers to implement snapshot on the application level [3, 25]. These technologies require modification of OS code or recompilation of applications, thus violating the transparency from the view of implementation schema.

The distributed snapshot of VMC seems to be an ef-

fective way to mitigate the transparency problem, since it implements snapshot on virtual machine manager (VMM) layer which encapsulates the application's running state and resources without modification to target applications or the OS. Many systems such as VNSnap [18] and Emulab [12] have been proposed to create the distributed snapshot for a closed network of VMs. However, these methods still have obvious shortcomings.

First, the snapshot should be non-disruptive to the upper applications, however the state-of-the-art VM snapshot technologies, either adopt stop-and-copy method (e.g., Xen and KVM) which causes the service are completely unavailable, or leverage live migration based schema which also causes long and unpredictable downtime owing to the final copy of dirty pages [26].

Second, the distributed snapshot should coordinate the individual snapshots of VMs to maintain a global consistent state. The global consistent state reflects the snapshot state in one virtual time epoch and regards causality, implying the VM before snapshot cannot receive the packets send from the VM that has finished the snapshot to keep the consistent state during distributed snapshot (further explanations about global consistent state can be referred in appendix A). However, due to the various VM memory size, variety of workloads and parallel I/O operations to save the state, the snapshot start time, duration time and completion time of different VMs are always different, resulting in the TCP back-off issue [18], thereby causing network interruption between the communicating VMs. Figure 1 demonstrates one such case happened in TCP's three-way handshake. Worse still, for the master/slave style distributed applications, the master always undertake heavier workloads so that cost more time to finish the snapshot than the slaves, therefore, the slaves which finish the snapshot ahead cannot communicate with the master until the master snapshot is over, causing the whole system hung. As a result, the master snapshot becomes the *short-board* during distributed snapshot of master/slave systems.

Third, most distributed snapshot technologies adopt the coordinated snapshot protocol [13] to bring the distributed applications into a consistent state. This requires a coordinator to communicate snapshot-related commands with other VMs during snapshot. In many systems, the coordinator is setup in the customized module such as VIOLIN switch in VNSnap [18] and XenBus handler used in Emulab [12], thus lack of generality in most virtualized environments.

To mitigate the problems above, we propose HotSnap, a system capable of taking *hot* distributed snapshot that is transparent to the upper applications. Once the snapshot command is received, HotSnap first suspends the VM, freezes the memory state and disk state, creates a *transient snapshot* of VM, and then resumes the VM. The

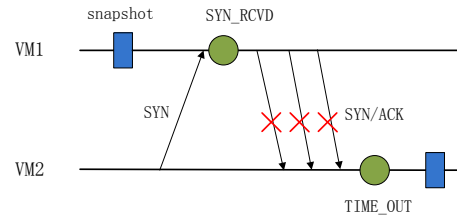


Figure 1: A TCP handshake case during distributed snapshot. VM_2 first sends SYN to VM_1 to request a TCP connection, at this moment VM_2 has not begin its snapshot; VM_1 receives this request, turn its own state into SYN_RCVD, and then sends SYN/ACK back to VM_2 . We notice that now VM_1 has finished snapshot, and based on the coordinated protocol, packets sent from VM_1 will not be accepted by VM_2 until VM_2 has finished its own snapshot. If VM_2 's snapshot duration exceeds TCP timeout, connection will fail.

transient snapshot only records the minimum instantaneous state, including CPU and device states, as well as two bitmaps reserved for memory state and disk state, bringing only milliseconds of VM downtime, i.e., *hot* for upper applications. The *full snapshot* will be acquired after resuming the VM, it saves the entire memory state in a copy-on-write (COW) manner, and create the disk snapshot in the redirect-on-write (ROW) schema; the COW and ROW schemas enable creating the *full snapshot* without blocking the execution of VM, i.e., live snapshot. Because the *transient snapshot* introduces only milliseconds of downtime, the discrepancy of downtime among different VM snapshots will be minor, thereby minimizing the TCP backoff duration.

HotSnap is completely implemented in VMM layer, it requires no modification to Guest OS or applications, and can work without other additional modules. The major contributions of the work are summarized as follows:

- 1) We propose a VM snapshot approach combined of *transient snapshot* and *full snapshot*. The approach completes snapshot transiently, enables all VMs finish their snapshots almost at the same time, which greatly reduces the TCP backoff duration caused by the discrepancy of VMs' snapshot completion times.
- 2) A classic coordinated non-blocking protocol is simplified and tailored to create the distributed snapshot of the VMC in our virtualized environment.
- 3) We implement HotSnap on QEMU/KVM platform [20]. Comprehensive experiments are conducted to evaluate the performance of HotSnap, and the results prove the correctness and effectiveness of our system.

The rest of the paper is organized as follows. The next section provides an analysis of the traditional dis-

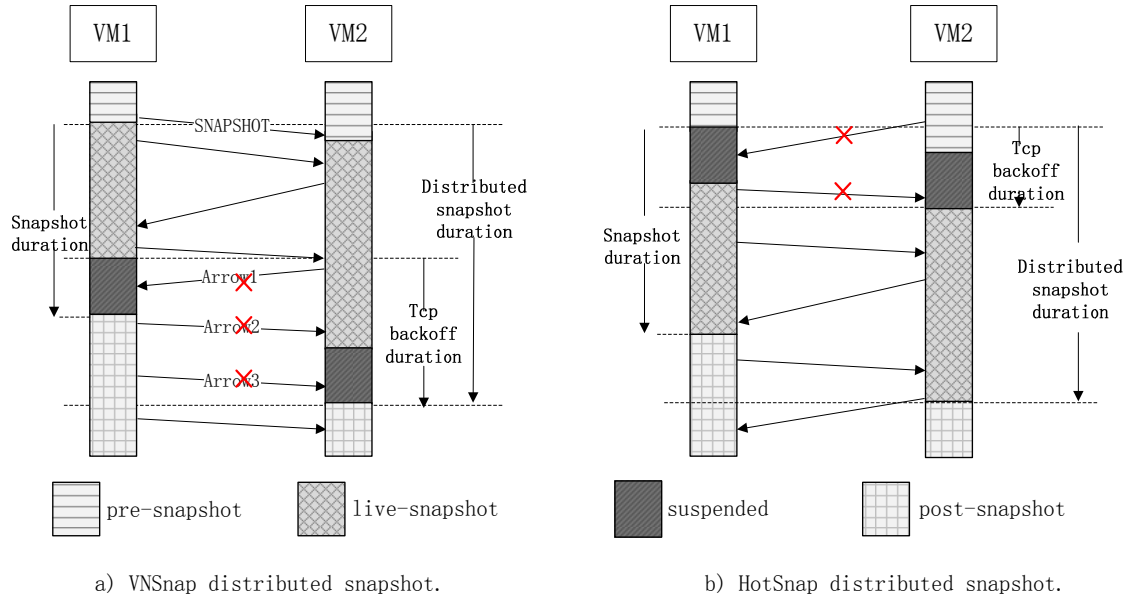


Figure 2: Comparison of VNSnap and HotSnap.

tributed snapshot and their problems. Section 3 introduces the HotSnap method, describes the *transient snapshot*, *full snapshot* and coordinated protocol. Section 4 describes the implementation-specific details on QEMU/KVM platform. The experimental results are shown in Section 5. Finally we present the previous work related to HotSnap in section 6 and conclude our work in Section 7.

2 An Analysis of Distributed Snapshot

The distributed snapshot includes independent VM snapshot and the coordinated protocol. Stop-and-copy schema is a simple way to create snapshot of individual VM, but this schema introduces long downtime of Guest OS and the upper applications running inside the VM, thus is impractical in many scenarios that deliver services to users. The live snapshot technologies leverage pre-copy based migration to achieve live snapshot by iteratively saving the dirty pages to the snapshot file [12, 18]. In this section, we will analyze the live migration based distributed snapshot proposed in VNSnap [18], and explain how it results in TCP backoff problem.

Figure 2(a) demonstrates the procedure of VNSnap distributed snapshot. Although VNSnap exploits the VIOLIN [12] switch to execute the coordinated protocol, we treat VM_1 as the coordinator for clarity. Upon distributed snapshot, the coordinator, i.e., VM_1 , will send SNAPSHOT command to VM_2 , and then create the snapshot of VM_1 itself. VNSnap leverages live migration to

iteratively save the dirtied pages into stable storage or reserved memory region until some requirements are satisfied, such as the amount of dirty pages are minor enough, or the size cannot be further reduced even more iterations are conducted. Then VNSnap suspends the VM, stores the final dirty memory pages, saves other devices' state and creates the disk snapshot. After these steps, the snapshot of VM_1 is over and VM_1 is resumed. Upon receiving the SNAPSHOT command from VM_1 , VM_2 follows the same procedure as VM_1 to create its own snapshot. VNSnap drops the packets send from the post-snapshot VM to pre-snapshot VM, to keep the global state consistent.

Take this tiny cluster which consists of two VMs as an example, the distributed snapshot duration time is from the start time of VM_1 snapshot to the end time of VM_2 snapshot (suppose VM_2 finishes snapshot later than VM_1), the TCP backoff duration is from the start of VM_1 suspend to the end of VM_2 suspend. The packets result in TCP backoff fall into three categories: 1) VM_1 is suspended while VM_2 is in live-snapshot, the packets send from VM_2 to VM_1 will not arrive, as *Arrow₁* illustrates; 2) VM_1 finishes snapshot and then turns into post-snapshot state, but VM_2 is before or during snapshot. In this situation, packets send from VM_1 will be dropped to keep the consistent state of distributed snapshot. *Arrow₂* shows such a case. 3) VM_1 is in post-snapshot, but VM_2 is suspended, VM_2 cannot receive the packets send from VM_1 , as *Arrow₃* shows.

Based on the three types of packets, we can conclude that two aspects affect the TCP backoff duration in dis-

tributed snapshot. One is the downtime of individual VM snapshot; the longer downtime implies more lost packets, thereby causing longer TCP backoff duration. Another is the discrepancy of the snapshot completion times, as the *Arrow₂* illustrates, the packets sent from *VM₁* which has finished snapshot ahead will be dropped until *VM₂* completes the snapshot.

According to the above analysis, the VNSnap distributed snapshot method has three drawbacks: First, the pre-copy based live migration method needs to iteratively save the dirtied pages, thus the snapshot downtime is directly related to the workloads inside the VM and I/O bandwidth; it may last seconds in memory intensive scenarios [26]. Second, VNSnap proposes a VNSnap-memory method to reduce the TCP backoff duration, it saves the memory state into a reserved memory region whose size is the same to the VM memory size; this is wasteful and impractical in the IaaS platform which aims to maximize resource utilization. Third, the snapshot duration time is proportional to the memory size and workload, therefore the discrepancy of snapshot completion times would be large for VMs with various memory sizes, further leads to long TCP backoff duration. Even for the VMs with identical memory size and same applications, the snapshot completion times are still various owing to the parallel disk I/O for saving large amount of memory pages. Besides, the experimental results in VNSnap [18] also show this live migration based snapshot method brings seconds of TCP backoff duration.

3 Design of HotSnap

The design of HotSnap includes a new individual VM snapshot method and a coordinated non-block snapshot protocol. We firstly describe the design of the HotSnap method, then introduce the procedure of HotSnap for individual VM, and lastly describe the coordinated snapshot protocol to acquire a global consistent state of the virtual machine cluster.

3.1 Overview of HotSnap

Figure 2(b) illustrates our HotSnap approach. Different from VNSnap which suspends the VM at the end of the snapshot, HotSnap suspends the VM once receiving the SNAPSHOT command, takes a *transient snapshot* of VM and then resumes the VM. The *full snapshot*, which records the entire memory state, will be completed during the subsequent execution. Note that the VM actually turns to post-snapshot state after *transient snapshot* is over. In this approach, the TCP backoff duration is from the start of *VM₁ transient snapshot* to the end of *VM₂ transient snapshot*, and the entire distributed snap-

shot duration starts from the start of *VM₁ transient snapshot* to the end of *VM₂ full snapshot*.

In HotSnap approach, we suspend the VM and create the *transient snapshot* in milliseconds, thus the downtime during individual VM snapshot would be minor. Besides, in the nowadays IaaS platform or data center which are always configured with high bandwidth and low latency network, the round trip time is always less than 1ms, so the VMs can receive the SNAPSHOT command and then start to create snapshot almost simultaneously. As a result, the *transient snapshot* of VMs can start almost simultaneously and finish in a very short time, consequently minimizing the TCP backoff duration.

The individual VM snapshot combined of *transient snapshot* and *full snapshot*, as well as the coordinated protocol are two key issues to create the *hot* distributed snapshot, and will be described in detail in the following parts.

3.2 Individual VM Snapshot

A VM snapshot is a point-in-time image of a virtual machine state; it consists of memory state, disk state and devices' states such as CPU state, network state, etc. Our snapshot consists of two parts, one is a *transient snapshot* which contains the devices state, disk state and metadata of memory state; another is *full snapshot* which actually records the memory state. We divide the individual VM snapshot procedure into three steps as shown in Figure 3.

Step 1, Suspend VM and Create Transient Snapshot. We suspend the VM, store the devices state, set write-protect flag to each memory page, create two bitmaps to index the memory state and disk state, and create a new null disk file. We adopt the redirect-on-write method to create disk snapshot, therefore the disk snapshot is completed after the bitmap and disk file are created. This step only involves lightweight operations, i.e., bitmap creation, flag setting and device state saving, thus bringing only a few dozens of milliseconds downtime.

Step 2, Resume VM and Create Full Snapshot. We resume the VM to keep the Guest OS and applications running during *full snapshot*. The running applications will issue disk writes as well as dirty memory pages during fault-free execution. For the disk write operation, the new content will be redirected to a new block in the new disk file by the iROW block driver [23]. For the write operation on one memory page which is write-protected, page fault will be caught in the VMM layer. HotSnap for handling page fault will block the memory write operation, store the original page content into the snapshot file, remove the write-protect flag, and then allow the guest to continue to write the new content into the page. Meanwhile, a thread is activated to save memory pages ac-

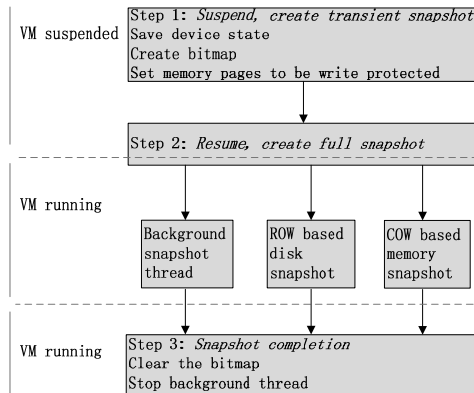


Figure 3: Steps of individual VM snapshot.

tively in background. The copy-on-write based memory snapshot only saves each memory page once and thus keeps the memory snapshot size to be the same to the VM memory size.

Step 3, Snapshot Completion. After all the memory pages are stored into the stable storage, the snapshot process is completed. In the end, we clear the bitmap, and stop the background thread.

3.3 Global Coordinated Non-block Snapshot Protocol

We design a global coordinated non-block snapshot protocol to coordinate the individual VM snapshot processes. Unlike Emulab [12] which synchronizes the clocks of all VMs to ensure these VMs are suspended for snapshot simultaneously, we deploy VMs on high bandwidth and low latency network so that the VMs can receive the message and start the snapshot at the same time. It is worth noting that Emulab's clock synchronization protocol can be utilized to extend the scope of HotSnap.

The pre-snapshot, live-snapshot and post-snapshot are both running state of individual VM, but they need to be distinguished from the view of VMC distributed snapshot for consistency requirement. Note that VNSnap suspends VM at the end of snapshot, so the live-snapshot can be regarded as pre-snapshot. Similarly, in HotSnap which suspends VM at the start, we consider the live-snapshot as post-snapshot, i.e., the state after *transient snapshot*. We leverage the message coloring [21] method to achieve the state distinction in the coordinated protocol, that is, we piggyback the white flag to the packet which is send from the VM in pre-snapshot state and represent the packets from the post-snapshot VM with red flag. If one pre-snapshot VM receives a packet piggybacked with a red flag, it will create its own snapshot first, and then receive and handle the packet.

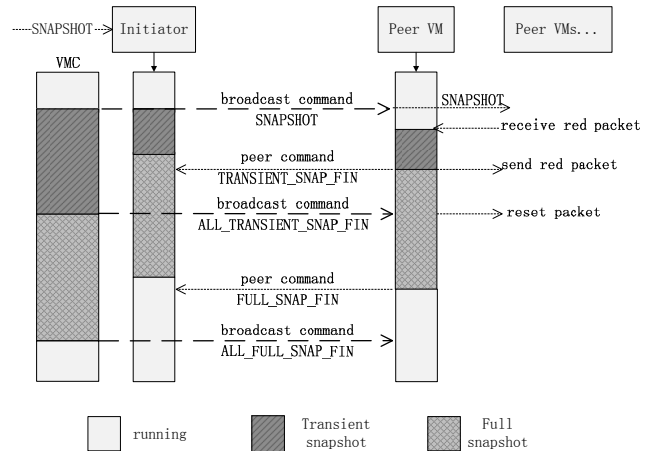


Figure 4: Global distributed snapshot protocol of HotSnap.

There exist two roles in HotSnap system: initiator and peer. Unlike the VNSnap [18] or Emulab [12] systems that use separate modules such as VIOLIN switch or XenBus as the initiator, each VM in HotSnap can be either the initiator or a peer. And there is only one initiator during a failure-free VMC snapshot process. Each peer records its snapshot states including *transient snapshot* state and *full snapshot* state. The initiator not only records the snapshot states, but also maintains these states of the whole VMC distributed snapshot. Figure 4 illustrates how the coordinated protocol works, the initiator after receiving the SNAPSHOT command from the user or administrator, will first broadcast this command to all peers, and then takes its own *transient snapshot*. The peers will trigger the snapshot process when receiving two kinds of packets, the SNAPSHOT message from the initiator, or the packet piggybacked with red flag. Once finishing the *transient snapshot*, the peer VM will send a TRANSIENT_SNAP_FIN message to the initiator, and color the transmitted packets with the red flag to imply the peer is in post-snapshot state. After finishing the snapshot itself and receiving all peers' TRANSIENT_SNAP_FIN messages, the initiator will broadcast the ALL_TRANSIENT_SNAP_FIN message to all peer VMs to notify the completion of *transient snapshot* of the whole VMC. The peers who receive this message will cancel packet coloring and reset the packet with the white flag immediately. The distributed snapshot procedure continues until the initiator receives all VMs' FULL_SNAP_FIN message which marks the completion of the *full snapshot*. The initiator will finally broadcast ALL_FULL_SNAP_FIN message to all peer VMs, to declare the completion of the distributed snapshot of the VMC.

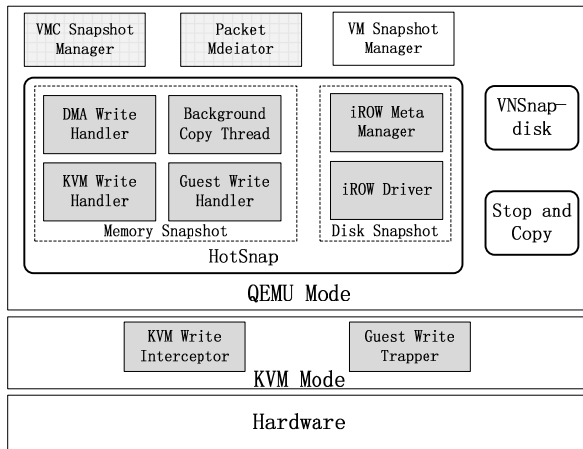


Figure 5: HotSnap system architecture.

4 System Implementation

This section presents the implementation issues in HotSnap. We start by describing the overall architecture, and then go on sub-level details and optimizations.

4.1 System architecture

We implement HotSnap on qemu-kvm-0.12.5 with Linux kernel 2.6.32.5-amd64. The system architecture is illustrated in Figure 5. HotSnap consists of two main components: the coordinated snapshot protocol component and VM snapshot component.

The coordinated protocol component includes the VMC Snapshot Manager and Packet Mediator. The VMC Snapshot Manager acts as the initiator during the distributed snapshot; it will firstly broadcast the SNAPSHOT command to other VMs, and then notify the VM Snapshot Manager to take the snapshot of VM itself. Packet Mediator has two functions: change the color of the sending packets according to the VM snapshot progress; decide whether or not to accept a packet by comparing packet's color with its own state.

VM Snapshot Manager is in charge of taking the individual VM snapshot; it supports three snapshot schemas, the HotSnap method, the Stop-and-copy snapshot method which is default adopted in QEMU/KVM, and the VNSnap-disk snapshot proposed in VNSnap [18] based on the pre-copy live migration in QEMU. In our HotSnap system, VM Snapshot Manager calls the Disk Snapshot Module to create the disk snapshot in the *transient snapshot* procedure, and exploits the Memory Snapshot Module to save the memory state during both *transient snapshot* and *full snapshot*. The details about Disk Snapshot Module can be referred in our iROW work

[23], thus are omitted in this paper.

4.2 COW Based Memory Snapshot

We create the memory snapshot in the copy-on-write (COW) manner. Writing large amount of memory state into a file within the kernel is terrible and may crash the system, so we save the memory state in user space when taking the snapshot. However, in the QEMU/KVM platform, the guest memory pages will be written by not only the guest OS or applications inside, but also by the simulated DMA devices and the KVM module. In the following, we will describe how we save the entire memory state in detail.

Guest Write Handler: During the *transient snapshot*, we call the function `cpu_physical_memory_set_dirty_tracking` in QEMU to set the write protect flag for each guest physical memory page, so that the VMM layer can trap the write page fault triggered by the running applications or the guest OS. Once page fault occurs, the execution of Guest OS will hang and exit to KVM module. Then we handle the page fault in the function `handle_ept_violation`, which is defined to handle memory access violation for the specific CPU with the Extended Page Tables (EPT) feature. If the trapped page fault is owing to be write protected, we record the guest frame number (gfn) of the page, set the exit reason as `EXIT_REASON_HOTSnap`, and then exit to QEMU for handling the exception. The QEMU component when encountering the exit whose reason is `EXIT_REASON_HOTSnap`, will save the memory page indexed by gfn into the snapshot file, notify KVM to remove the write protect flag of the page, and then issue the function `kvm_run` to activate the VM. Afterwards, the resumed VM will continue to run and write that memory page without triggering page fault again. In addition, each page fault incurs exit from Guest to KVM kernel then to QEMU user space, as well as entry in the opposite direction, resulting in performance loss. Thus, we save dozens of memory pages and remove their associated write protect flags when handling one page fault. This can benefit from the feature of memory locality, thereby reducing the frequency of page fault and occurrences of context switch between Guest OS and VMM layer.

DMA Write Handler: DMA is a widely-used technology to accelerate the transfer speed of I/O related devices. QEMU also simulate the DMA schema for IDE driver and virtio network driver in qemu-kvm-0.12.5. In their implementation, a reserved memory region is mapped between the guest and the simulated DMA driver. For read or write operations, the driver just map the data to or from the guest directly instead of I/O operations. This method dirties the guest memory pages with-

out triggering the page fault and thus cannot be caught in the Guest Write Handler way. Therefore, we intercept the DMA write operations directly in QEMU. Take disk I/O as an example, the function *dma_bdrv_cb* in QEMU implements the DMA write operations, it first fetches the address and length of the data to be written from the scatter/gather list, maps this address to the guest physical page by the function *cpu_physical_memory_map* and finally writes the data into the guest disk file. Therefore, we intercept the write operation in the function *dma_bdrv_cb*, save the original memory page content, and then resume the execution of DMA write. One thing to be noted is that only the disk device and network device in qemu-kvm-0.12.5 support the DMA schema, but the newer versions such as qemu-kvm-1.4.0 add the DMA feature to more devices including disk device, sound card and video card. However, the methodologies are the same in dealing with DMA interception.

KVM Write Handler: The KVM kernel set value to the registers such as MSR for task switch operations, key-board operations, thus causing the guest memory pages dirtied. Similar to DMA Write Handler, we intercept KVM write in the function *kvm_write_guest* which is implemented in KVM to assign value to certain address space. The KVM kernel always repeatedly writes the same memory pages and the page count is only a little, so we first store the intercepted pages into a kernel buffer without blocking the execution of *kvm_write_guest*, then copy these pages in buffer to the user space asynchronously. In this way, all the memory pages dirtied by KVM will be saved to stable storage in the user space.

Background Copy: To accelerate creating the memory snapshot, a background copy thread is issued to store the guest memory pages concurrently. It traverses all the guest memory pages and saves the pages that have not been saved by the other three memory snapshot manners. Since all these four kinds of memory snapshot manners may save the same page simultaneously, a bitmap is reserved for indexing whether the page is saved or not, to guarantee completeness and consistency of the memory state. The bitmap is shared between QEMU and KVM. All the four manners should first store the page, then set the associated bit value and remove the write-protect flag. Or else, there will be concurrency bugs. Let us consider an improper case that set bit and removes flag first. Upon saving a page, the Background copy thread firstly set the associated bit and removes the write-protect flag; however, before the thread saving the page content, the Guest OS dirties this page since write protect flag has been removed, causing the thread to save the false page content. Thus, when taking memory snapshot, the interceptions on DMA write and KVM write will first check the associated bit value of the page about to write, save

the original page if the bit value is not set, or ignore otherwise. The Guest Write Handler also takes the same procedure, for the bit value has been set, it allows the guest to continue running without exit to QEMU.

4.3 Log and Resend On-the-fly Packets

Dropping the on-the-fly packets send from post-snapshot VM to pre-snapshot VM during TCP backoff duration is a simple way to obtain the global consistent state. However, this way will increase the TCP backoff duration, the reason is as follows: TCP or other upper level protocols will retransmit the packets that are not acknowledged in a certain time named Retransmit Timeout (RTO), to achieve correctness and reliability of message passing. That means, if the packets are lost, these reliable protocols will delay resending the packet until timeout. The default RTO value is 3 seconds in Linux 2.6.32 kernel, it is always larger than the TCP backoff duration which lasts tens of milliseconds in HotSnap system (in Section 5.3). Thus, if the packets are dropped, the actual network interruption time will be the RTO value at the minimum, i.e., 3 seconds. Worse still, the RTO value will increase manyfold until receiving the acknowledgement.

Instead of dropping the packets directly, the Packet Mediator component intercepts the read/write operations issued by the tap device which is connected to the virtual network interface card (VNIC) of VM, logs the on-the-fly packets send from the post-snapshot VM to the pre-snapshot VM, and then stores the packets into a buffer. After completing the *transient snapshot*, the Packet Mediator will first fetch the packets from the buffer, send them to the VNIC of the VM, and finally resume the normal network communication.

5 Experimental Evaluation

We apply several application benchmarks to evaluate HotSnap. We begin by illustrating the results for creating snapshot of individual VM, and then compare the TCP backoff duration between three snapshot modes under various VMC configurations, lastly we characterize the impacts on performance of applications in VMC.

5.1 Experimental Setup

We conduct the experiments on four physical servers, each configured with 8-way quad-core Intel Xeon 2.4GHz processors, 48GB DDR memory, and Intel 82576 Gigabit network interface card. The servers are connected via switched Gigabit Ethernet. We configure 2GB memory for the virtual machines unless specified otherwise. The operating system on physical servers and

virtual machines is debian6.0 with 2.6.32-5-amd64 kernel. We use qemu-kvm-0.12.5 as the virtual machine manager. The workloads inside the VMs includes:

Idle workload means the VM does nothing except the tasks of OS self after boot up.

Kernel Compilation represents a development workload involves memory and disk I/O operations. We compile the Linux 2.6.32 kernel along with all modules.

Matrix Multiplication multiplies two randomly generated square matrices, this workload is both memory and CPU intensive.

DBench [4] is a well known benchmark tool to evaluate the file system, it generates I/O workloads.

Memcached [8] is an in-memory key-value store for small chunks of data, the memcached server when receiving a request containing the key, will reply with the value. We set memcached server in one VM, and configure mcblaster [7] as client in another VM to fill the data in the memcached instance and then randomly request the data from the memcached server.

Distcc [5] is a compilation tool that distributes the compilation tasks across the VMs connected in the VMC. It contains one Distcc client and several servers. The client distributes the tasks to servers, and the servers after receiving the task will handle the task and then return the result to the client. This workload is memory and network intensive. We use Distcc to compile the Linux 2.6.32 kernel with all modules in the VMC.

BitTorrent [2] is a file transferring system, the peers connect to each other directly to send and receive portions of file. Different from distcc that is centralized, BitTorrent is peer-to-peer in nature.

We compare the three snapshot methods, all these methods save the snapshot file in local host.

Stop-and-copy. The default snapshot method used in QEMU/KVM, it suspends the VM while creating snapshot.

VNSnap-disk. We implement the VNSnap-disk snapshot method based on live migration in QEMU/KVM, save the memory state into the stable storage directly.

HotSnap. Our snapshot method that suspends the VM first, then create the *transient snapshot* and *full snapshot*.

QEMU/KVM optimizes taking snapshot by compressing the zero pages with one byte, thus reduce the amount of saved state. This incurs unfairness in experiments, the reason is, the VM after long time running may dirty more zero pages, and experience longer snapshot duration than the new booted VM which contains large number of zero pages, thus leading to unpredictable TCP backoff duration between the two VMs. As a result, we abandon the compression codes, save the whole zero page instead of only one byte to eliminate the impact of zero pages.

5.2 Snapshot of Individual VM

We start by verifying the correctness of saved snapshot state for individual VM, and then evaluate the snapshot metrics including downtime, duration and snapshot file size. When calculating the snapshot file size, we count all devices' state, memory state and bitmaps, as well as the disk snapshot which is either a bitmap file in HotSnap or multi-level tree file in the other two modes.

Correctness: Correctness means the snapshot correctly records all the state of the running VM, so that the VM can rollback to the snapshot time point and continue to run successfully. To verify the correctness of HotSnap, we compile the Linux kernel and take several snapshots during normal execution. We pick the snapshots and continue running from these snapshot points, the compiled kernel and modules can execute successfully. Besides, we take snapshot in the Stop-and-copy manner, and then create the HotSnap snapshot. The content of the two snapshots are identical and thus demonstrate the correctness of our system.

Snapshot Metrics: We run Kernel Compilation, Matrix Multiplication, Memcached and Dbench applications to evaluate the performance when taking snapshot of individual VM. We compare HotSnap with Stop-and-copy and VNSnap-disk in terms of snapshot duration, downtime and snapshot file size. As shown in Table 1, the VM only experiences about 35 milliseconds downtime during HotSnap, because this step is to create the *transient snapshot* which only involves lightweight operations. The downtime in VNSnap-disk is various, e.g., 381ms for kernel compilation and 36.8ms when idle, it is related to the workload and I/O bandwidth. HotSnap also achieves shorter snapshot duration and smaller file size than VNSnap-disk. This is because HotSnap saves only one copy for each memory page, while the live migration based VNSnap-disk needs to iteratively save dirty memory pages to snapshot file. The Stop-and-copy method, obviously, incurs dozens of seconds downtime. The bitmap file size is only a little, e.g., 128KBytes to index 4GBytes memory of VM, so that the snapshot file size in HotSnap and Stop-and-copy are both about 2.02GBytes.

5.3 Snapshot of VMC

In this section, we evaluate HotSnap in the virtual machine cluster and focus on the TCP backoff duration. We compare the TCP backoff duration in three snapshot modes, while changing the VMC configurations. The VMC configurations include: VMC under various workload, VMC of different scales, VMC with different VM memory size and disk size, and VMC mixed of VMs with different memory size.

We will first illustrate the details on snapshot progress

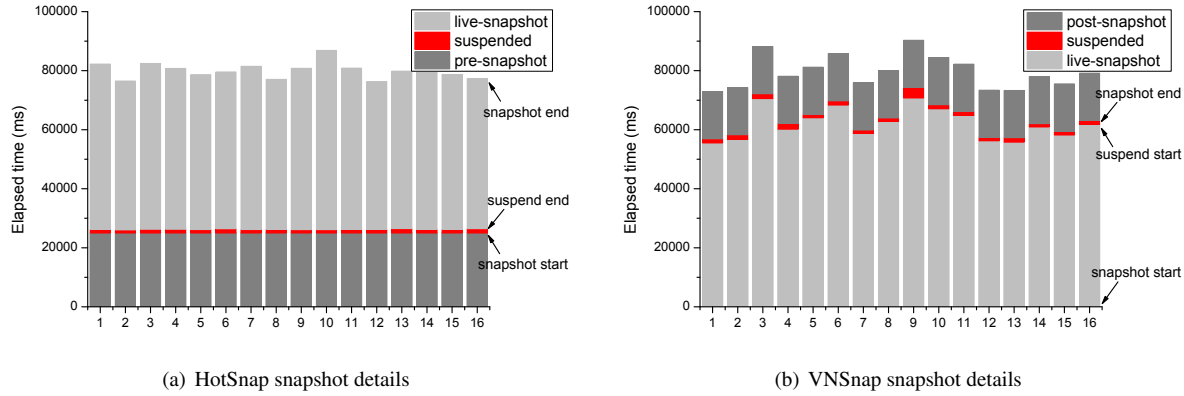


Figure 6: Comparison of TCP backoff details.

that lead to TCP backoff. We build a VMC with 16 VMs, with Distcc running inside. When creating the individual VM snapshot, we record the snapshot start time, VM suspend start time, VM suspend end time and snapshot completion time. The suspend end time equals snapshot completion time in VNSnap-disk snapshot method because VNSnap-disk suspends the VM at the end of snapshot, the TCP backoff duration between two VMs is from the first VM suspend start time to the last VM snapshot completion time. The snapshot start time is identical to the suspend start time in HotSnap because HotSnap suspends the VM at the start of snapshot, and the TCP backoff duration is from the suspend start time to the last suspend end time. Figure 6 shows the detailed results of 16 individual VM snapshot progresses in the VMC. The Stop-and-copy downtime last dozens or even hundreds of seconds, thus are not given in the figure.

We can see from Figure 6(a) that the duration from snapshot start to suspend end of VMs are almost identical and are minor, and the average suspend duration (downtime) is 116ms. The maximum TCP backoff duration between two VMs is 273ms. For VNSnap-disk snapshot method shown in Figure 6(b), although the snapshots s-

tart simultaneously, their suspend start time are various owing to iteratively saving the dirtied memory pages. The suspend duration are also different, ranges from tens of milliseconds to hundreds of milliseconds. The VM_9 which is the Distcc client even suffers from 2.03 seconds downtime, because it undertakes the heaviest task and generates large amount of memory during final transfer. VNSnap-disk brings 359ms VM downtime in average, only a little more than that of HotSnap. However, due to the discrepancy of snapshot completion times, the TCP backoff duration is much longer, e.g., the maximum value is 15.2 seconds between VM_1 and VM_9 . This result suggests that the TCP backoff in VNSnap-disk snapshot method is much severe in the master/slave style distributed applications. The master always suffers from heavier workloads, costs longer time to finish the snapshot than the slaves, resulting in longer network interruption between master and slaves. However, the HotSnap method can effectively avoid this short-board affect because the downtime to create the *transient snapshot* is regardless of the workload, and lasts only tens of milliseconds.

The TCP backoff duration between two VMs is easy to acquire, but the backoff duration for the whole VMC

Metrics	Duration(s)			Downtime(ms)			Snapshot Size(GBytes)		
Benchmarks	Stop-and-copy	VNSnap-disk	HotSnap	Stop-and-copy	VNSnap-disk	HotSnap	Stop-and-copy	VNSnap-disk	HotSnap
Idle	50.64	51.66	51.57	50640	36.83	31.88	2.02	2.04	2.02
Compilation	52.50	61.11	51.96	52500	381.72	34.16	2.02	2.38	2.02
Matrix Multiplication	49.34	51.75	52.31	49340	55.73	35.93	2.02	2.19	2.02
Memcached	53.09	69.43	54.72	53090	150.85	33.80	2.02	2.41	2.02
Dbench	56.93	60.76	50.18	56930	79.36	39.36	2.02	2.17	2.02

Table 1: Comparison of individual VM snapshot techniques.

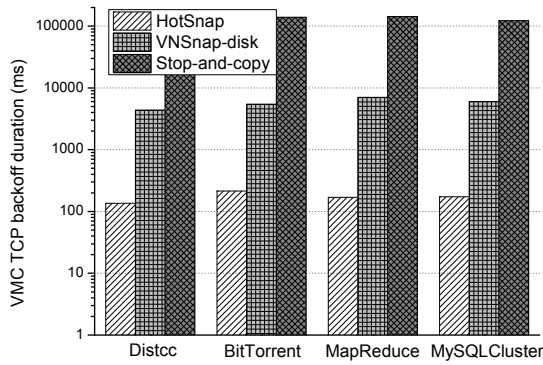


Figure 7: TCP backoff in VMC under various workloads.

is hard to depict, we approximate the value by the average of TCP backoff durations between each two VMs in the VMC. Take Figure 6 as an example, the VMC TCP backoff duration are 137ms and 8.6s for HotSnap and VNSnap-disk methods respectively. In the following, we will compare the results of VMC TCP backoff duration under different VMC configurations in Figures 7-10. The VMC TCP backoff duration is log transformed on y-axis for clear comparisons:

VMC under various workloads. In this configuration, we build the VMC with 16 VMs, which are deployed evenly on four physical servers. The workloads are Distcc, BitTorrent, MapReduce and MySQL Cluster. For BitTorrent, we set one tracker in one VM, set two VMs as clients to download files from other VMs as seeds. We set up the Hadoop MapReduce [6] to count the key words in the short messages data set from China unicom, which contains over 600 million messages. Besides, we use the MySQL Cluster [9] to build a distributed database across the VMs, we configure one VM as management node, 13 VMs as database nodes, and exploit Sysbench [10] set up in two VMs to query the data in parallel. Figure 7 compares the VMC TCP backoff duration under different snapshot modes. As expected, the HotSnap distributed snapshot only suffers from about 100 milliseconds backoff duration under all the workloads, while the VNSnap-disk method incurs as many as 7 seconds in the MapReduce and MySQLCluster workloads, owing to the different snapshot completion times among VMs.

VMC of different scales. We set up the VMC with 8, 16, 24 and 32 VMs with Distcc running inside. Same as above, the VMs are deployed evenly on four physical servers. Figure 8 shows that the VMC TCP backoff duration in HotSnap method keeps almost constant, i.e., less than 200ms regardless of the number of VMs in the VM-

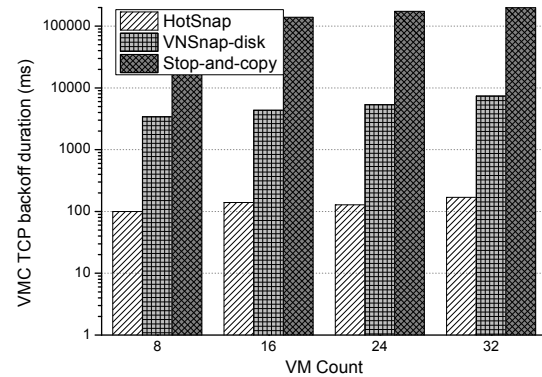


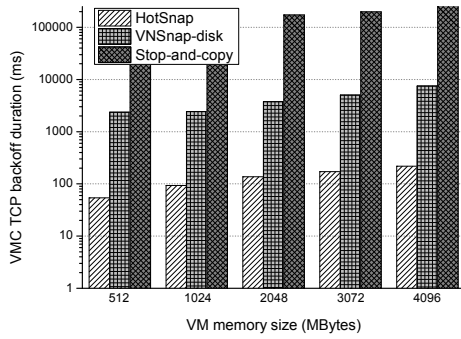
Figure 8: TCP backoff in VMC with different scales.

C. That's because the *transient snapshot* in HotSnap only save a few megabytes data, involves CPU state, bitmap files of disk and memory state. The VMC TCP backoff duration in VNSnap-disk and Stop-and-copy rises with the increase of VM number; the reason is the parallel execution of writing more large files.

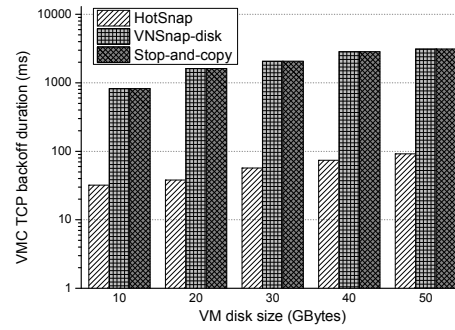
VMC with increased VM memory and disk size.

We configure the VMC with 16 VMs under the Distcc workload. Figure 9 compares the VMC TCP backoff duration while increasing the VM memory size and disk size. The VMC TCP backoff duration in all the three modes increase while increasing the VM memory size and disk size. The increase in HotSnap method is because HotSnap need to reserve larger bitmap files for larger disk and memory size, and more operations to set write protect flags to memory pages. The increase in Figure 9(a) in VNSnap-disk may come from two aspects: the first is the parallel execution of writing larger files, which is also the reason for the increase in the Stop-and-copy method; the second is owing to longer time to save more memory pages, which will further generate more dirtied pages during iteration time. Because the disk snapshot is created in VM suspend phase, and different disk size affects the VM downtime, so we show the downtime instead of VMC TCP backoff duration in Figure 9(b). As estimated, the redirect-on-write based snapshot method cost only tens of milliseconds, achieves the reduction by more than 20x compared to the other two modes. Besides, the VNSnap-disk downtime reaches several seconds, and is proportional to the VM disk size, because most of the downtime is consumed to index the disk blocks in the multi-level tree structure. The Stop-and-copy takes the same method to create disk snapshot, thus incurs the same downtime to VNSnap-disk.

VMC mixed of VMs with different memory size. In this VMC configuration, we set up the VMC with two



(a) TCP backoff in VMC with increasing VM memory size



(b) Downtime in VMC with increasing VM disk size. The disk is filled with data

Figure 9: TCP backoff in VMC with different VM configurations.

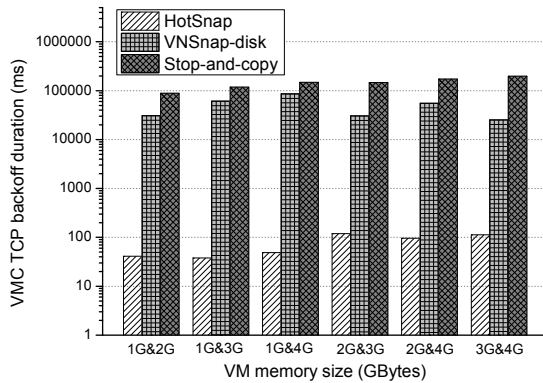


Figure 10: TCP backoff in VMC mixed of VMs with different memory size.

VMs of different memory size. As illustrated in Figure 10, compared to the HotSnap TCP backoff duration that keeps almost constant regardless of the discrepancy of VM memory size, the TCP backoff duration of VNSnap-disk increases proportionally to the raised discrepancy between memory size, i.e., every 1G memory difference will incur additional 27 seconds backoff duration. As expected, the VMC which consists of 1G memory VM and 4G memory VM obtains the largest backoff duration. The Stop-and-copy method, on the other hand, increases with the increasing memory size of VMs, this is easily understand because more time will be consumed to write larger files into the snapshot file.

5.4 Performance Impact on Individual VM

In the above experiments, we save the whole zero page instead of only one byte to avoid the impacts of zero pages. However, saving multiple large snapshot files simultaneously during distributed snapshot will degrade performance seriously, and even cause write operation timeout for I/O intensive applications. We consider the optimization as our future work, but in this paper, we simply resume the zero page compression mode to reduce the saved page count during snapshot.

As stated in the previous section, during the *full snapshot* step, we trap the write page fault in KVM and turn to QEMU to handle the write fault, so that the memory operations of Guest OS and user applications are affected. Besides, we intercept the DMA write operations, which may affect the guest I/O speed. So we first give the statistic on the four memory page saving manners during HotSnap, and then evaluate the performance on applications.

Page count of different manners in HotSnap. HotSnap saves the memory pages in four manners: Guest Write Handler, DMA Write Handler, KVM Write Handler and Background Copy. The saved page number of these four types under various workloads are listed in Table 2. The Guest Write pages always account more than that of DMA Write and KVM Write, but the amount is still minor even under memory intensive workloads, e.g., 2.5% of all memory pages under Memcached. This is possible because HotSnap saves dozens of neighbouring pages when handling one page fault and thus it benefits from the memory locality feature. Handling one Guest Write page cost about 60us, including the time to trap write page fault, exit to QEMU to save memory pages, remove write protect flag and resume the execution. As a result, the total cost incurred by saving

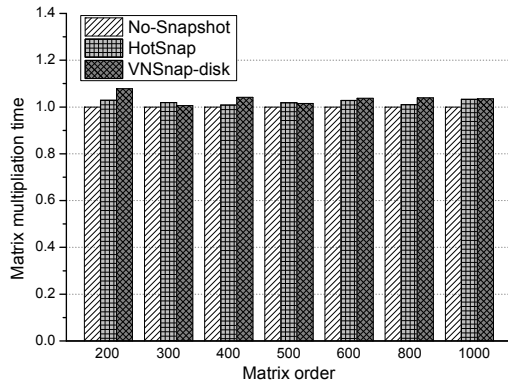


Figure 11: Matrix multiplication calculation time.

page in Guest Write is affordable. DMA Write operations and KVM Write operations always repeatedly access the same memory pages, therefore the count is minor. Besides, the interception on DMA write and KVM write lasts about 12us, making the impact can be negligible.

Workloads	Background Copy	Guest Write	DMA Write	KVM Write
Idle	528301	133	44	2
Compilation	524506	3912	59	3
Matrix Multiplication	520496	7916	65	3
Memcached	514814	13270	394	2
Dbench	526831	987	660	2

Table 2: Count of four page types during HotSnap.

Matrix multiplication time. Matrix Multiplication involves large amount of memory and CPU operations. We calculate the multiplication of two matrices while increasing the matrix order, and obtain the calculation time during No-Snapshot (i.e., normal execution), HotSnap and VNSnap-disk snapshot. Figure 11 compares the results of HotSnap and VNSnap-disk to the completion time of No-Snapshot as baseline. Both the two live snapshot methods bring less than 5% additional time to finish the computation, implying no obvious performance penalty during distributed snapshot for this kind of workload.

Kernel compilation time. Kernel compilation involves both memory operations and disk IO operations, we compile Linux-2.6.32.5 kernel with all modules during continuous VM snapshot. Figure 12 compares the compilation duration in No-Snapshot, HotSnap and VNSnap-disk modes. The time during Hot-

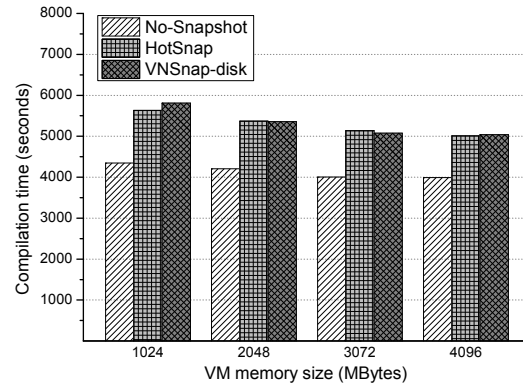


Figure 12: Kernel compilation time.

Snap and VNSnap-disk distributed snapshot are almost equal, and both consume about 20% more time to compile the kernel. The increase is owing to the CPU utilization and I/O bandwidth consumed by the VM snapshot. The 20% reduction maybe unacceptable in many performance-critical systems, and we leave the performance optimization as our future work.

5.5 Performance Impact on VMC

We first build the VMC with different number of virtual machines, and run Distcc to compare the completion time of normal execution, HotSnap distributed snapshot and VNSnap-disk snapshot. Then we set the VMC with 16 virtual machines running on four physical servers evenly, and install BitTorrent to evaluate the download speed during distributed snapshot.

Distcc compilation time. Distcc client distributes the compilation task to servers, if it loses connection with one server, the client will do the task in local; and the client can continue to distribute the task once the server is connected again. Figure 13 depicts the compilation time during continuous distributed snapshot while increasing the number of VMs in the VMC. Compared to the No-Snapshot mode, the compilation duration during HotSnap distributed snapshot increases by about 20%. The increase are mainly due to the snapshot overhead such as I/O operations and CPU utilization, the similar results are also illustrated in Figure 12. The duration during VNSnap-disk is much longer, it cost about 7% to 10% more time to finish compilation than that of HotSnap. Obviously, this is due to the TCP backoff which incurs network interruption between client and servers.

BitTorrent download speed. We set up the BitTorrent to evaluate the network performance loss incurred during distributed snapshot. We build the tracker on

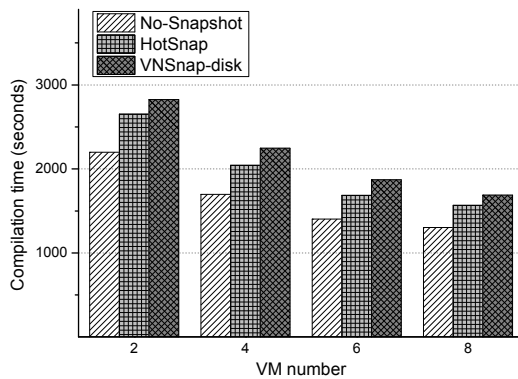


Figure 13: Distcc compilation time.

one VM, treat one VM as the client to download a large video file from other VMs as seeds, and record the download speed every 100 milliseconds. Note that the practical download speed varies significantly even in adjacent epoches, we average the speed by twenty samples. Figure 14 compares the download speed between normal execution and distributed snapshot. The download speed reduces from 33.2MBytes/sec during normal running to 22.7MBytes/sec when taking snapshot. The difference of impact between HotSnap and VNSnap-disk is also illustrated in the figure. HotSnap incurs a sharp decrease at about the 6th seconds, which is actually the distributed snapshot downtime to create the *transient snapshot*. Then the download speed will reach about 22.7MBytes/sec and keep the speed until the 55th second. From this time point, many VMs finish the *full snapshot* and resume to normal execution, so that the download speed will increase and finally reach the normal download speed, i.e., about 33.2MBytes/sec. The download speed during VNSnap-disk distributed snapshot shows opposite result from the 55th second, it decreases and reaches 0MBytes/sec at the 68th second. The reason is that the BitTorrent client experiences about 65 seconds to finish the snapshot, and will not receive the packets from seed VMs that finish the snapshot ahead, therefore decrease the download speed. After the client resumes the execution from the 71st second, the download speed also returns to normal.

6 Related Work

Distributed snapshot has been widely studied in the past thirty years, and many techniques have been proposed to create snapshot for distributed systems. The earlier works mainly focus on designing the snapshot protocol between the peers to maintain a global consistent state.

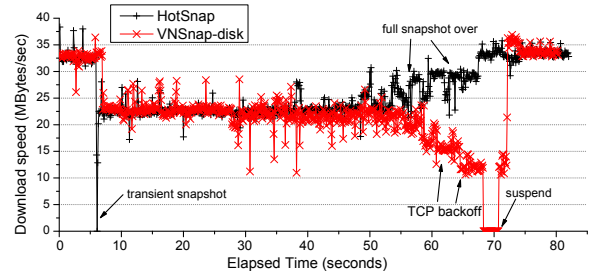


Figure 14: BitTorrent download speed.

Chandy and Lamport [13] assume the message channel is reliable and FIFO, and propose the coordinated protocol. Lai [21] proposes the message coloring based protocol to achieve consistency for non-FIFO channel. Kim [19] blocks the peer running until all the distributed snapshot finishes to reserve the consistent state. In contrast to optimize the snapshot protocol, we implement the coordinated, non-blocking protocol in the high bandwidth and low latency local area network, the protocol is simple and suitable for taking the distributed snapshot in the virtualized environments.

Another key aspect of distributed snapshot is the snapshot technology. Copy-on-write, redirect-on-write, split mirror are common methods to save the system state, and are implemented on kernel level [11], library level [24, 17, 25] or application level [3, 16] to create the snapshot. However, these methods require modification of the OS kernel or applications, thus is unacceptable in many scenarios.

The virtualization technology encapsulates the whole application as well as the necessary resources; it contributes to create the snapshot in an transparent manner. Emulab [12] leverages live migration technique to enable taking snapshots of the virtual machine network. By synchronizing clocks across the network, Emulab suspends all nodes for snapshot near simultaneously, and implements a coordinated, distributed snapshot. The synchronization will block the execution of the VMs, thus interfere with the applications running in the VMs. Besides, Emulab requires modifications to the Guest OS, and is hard to support legacy and commodity OS. VNSnap [18] also leverages Xen live migration [14] function to minimize system downtime when taking VMN snapshots. Unlike Emulab, VNSnap employs non-blocking coordination protocols without blocking VMs, and VNSnap requires no modification to the guest OS. Our HotSnap proposal shares a similar manner to VNSnap, but we design a *transient snapshot* manner to suspend the VM first and then create the *full snapshot*, which reduces the discrepancy of snapshot completion time. Besides, we log and resend the packets send from post-snapshot VM to pre-

snapshot VM instead of dropping them which is adopted by VNSnap. Both these two technologies achieve the notable reduction in the TCP backoff duration. Moreover, we treat one VM as the initiator to avoid setting up a customized module, which makes HotSnap to be easily portable to other virtualized environments.

7 Conclusions

This paper presents a distributed snapshot system HotSnap, which enables taking *hot* snapshot of virtual machine cluster without blocking the normal execution of VMs. To mitigate TCP backoff problem and minimize packets loss during snapshots, we propose a transient VM snapshot approach capable of taking individual VM snapshot almost instantaneously, which greatly reduces the discrepancy of snapshot completion times. We have implemented HotSnap on QEMU/KVM platform, and conduct several experiments. The experimental results illustrate the TCP backoff duration during HotSnap distributed snapshot is minor and almost constant regardless of the workloads, VM memory size and different VM-C configurations, thus demonstrate the effectiveness and efficiency of HotSnap.

There still exists several limitations in HotSnap. First, the newer QEMU version supports more DMA simulators such as sound card and video card, implementing DMA write interceptions for each simulated device are fussy. Second, creating distributed snapshot involves large amount of I/O operations, thus affect the I/O intensive applications running inside the VMs. Therefore, our ongoing works include designing an abstract layer to intercept DMA write operations, scheduling the I/O operations from Guest OS and HotSnap for applications' performance requirements. We also plan to evaluate HotSnap under real-world applications.

Acknowledgements

We thank Hanqing Liu, Min Li for their contributable work on disk snapshot. We thank our shepherd, Marco Nicosia, and the anonymous reviewers for their valuable comments and help in improving this paper. This work is supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302602, National High Technology Research 863 Program of China under Grant No. 2011AA01A202, National Nature Science Foundation of China under Grant No. 61272165, No. 60903149, No. 91118008, and No. 61170294.

References

- [1] Amazon elastic compute cloud (amazon ec2). [Http://aws.amazon.com/ec2/](http://aws.amazon.com/ec2/).
- [2] Bittorrent. <http://www.bittorrent.com/>.
- [3] Ckpt library. <http://pages.cs.wisc.edu/~zandy/ckpt/>.
- [4] Dbench. <http://dbench.samba.org/>.
- [5] Distcc. <http://code.google.com/p/distcc/>.
- [6] Mapreduce. <http://hadoop.apache.org>.
- [7] Mcblaster. <https://github.com/fbmarc/facebook-memcached-old/tree/master/test/mcblaster>.
- [8] Memcached. <http://memcached.org/>.
- [9] Mysql cluster. <http://dev.mysql.com/downloads/cluster/>.
- [10] Sysbench. <http://sysbench.sourceforge.net/docs/>.
- [11] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. A survey of rollback-recovery protocols in message-passing systems. *ACM Transactions Computer Systems*, 7(1):1–24, 1989.
- [12] A. Burtsev, P. Radhakrishnan, M. Hibler, and J. Lepreau. Transparent checkpoints of closed distributed systems in emulab. In *Proceedings of EuroSys*, pages 173–186, 2009.
- [13] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst*, 3(1):63–75, 1985.
- [14] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of NSDI*, pages 273–286, 2005.
- [15] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3): 375–408, 2002.
- [16] A. Goldberg, A. Gopal, K. Li, R. Strom, and D. F. Bacon. Transparent recovery of mach applications. In *Proceedings of Usenix Mach Workshop*, pages 169–184, 1990.
- [17] J. Hursey, J. M. Squyres, T. Mattox, and A. Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for open mpi. In *Proceedings of IPDPS*, pages 1–8, 2007.

- [18] A. Kangarlou, P. Eugster, and D. Xu. Vnsnap: Taking snapshots of virtual networked environments with minimal downtime. In *Proceedings of DSN*, pages 534–533, 2009.
- [19] J. Kim and T. Park. An efficient protocol for checkpointing recovery in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):955–960, 1993.
- [20] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. Kvm: the linux virtual machine monitor. *Computer and Information Science*, 1:225–230, 2007.
- [21] T. H. Lai and T. H. Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, 1987.
- [22] J. Li, B. Li, T. Wo, C. Hu, J. Huai, L. Liu, and K. P. Lam. Cyberguarder: A virtualization security assurance architecture for green cloud computing. *Future Generation Computer Systems*, 28:379–390, 2.
- [23] J. Li, H. Liu, L. Cui, B. Li, and T. Wo. irow: An efficient live snapshot system for virtual machine disk. In *Proceedings of ICPADS*, pages 376–383, 2012.
- [24] C. Ma, Z. Huo, J. Cai, and D. Meng. Dcr: A fully transparent checkpoint/restart framework for distributed systems. In *Proceedings of CLUSTER*, pages 1–10, 2009.
- [25] J. F. Ruscio, M. A. Heffner, and S. Varadarajan. Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems. In *Proceedings of IPDPS*, pages 1–10, 2007.
- [26] X. Song, J. Shi, R. Liu, J. Yang, and H. Chen. Parallelizing live migration of virtual machines. In *Proceedings of VEE*, pages 85–96, 2013.
- [27] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

A Global Consistent State

The virtual machine cluster is a message-passing system; therefore the global state of VMC to be saved consists of the individual states of single VMs and the states of the communication channels. The global consistent state requires: 1) if the state of a process reflects a message receipt, then the state of the corresponding sender must reflect sending this message; 2) A packet can be in the sender, or flying in the channel, or is accepted by the receiver, but cannot exist in two at the same time. Figure 15 illustrates a consistent and inconsistent state [15]. Figure 15(a) is consistent even the P_1 do not receive m_1 , because m_1 has been sent from P_0 , and is travelling in the channel in this case. On the other hand, Figure 15(b) describes an inconsistent state, this is because m_2 received by P_2 has not been sent from P_1 in this snapshot state. In such a case, P_1 will resend

m_2 after rollback to the inconsistent snapshot state, thus result in fault state of P_2 . As a result, this kind of messages that send from post-snapshot process to pre-snapshot process are always dropped, or logged and then resend after snapshot is over.

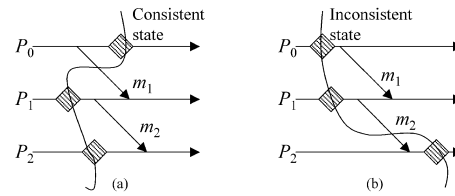


Figure 15: An example of consistent and inconsistent state. p stands for independent process, and m represents the message.

Supporting Undoability in Systems Operations

Ingo Weber^{1,2}, Hiroshi Wada^{1,2}, Alan Fekete^{1,3}, Anna Liu^{1,2}, and Len Bass^{1,2}

¹NICTA, Sydney

²School of Computer Science and Engineering, University of New South Wales

³School of Information Technologies, University of Sydney

¹{*firstname.lastname*}@nicta.com.au

Abstract

When managing cloud resources, many administrators operate without a safety net. For instance, inadvertently deleting a virtual disk results in the complete loss of the contained data. The facility to undo a collection of changes, reverting to a previous acceptable state, is widely recognized as valuable support for dependability. In this paper, we consider the particular needs of the system administrators managing API-controlled resources, such as cloud resources on the IaaS level. In particular, we propose an approach which is based on an abstract model of the effects of each available operation. Using this model, we check to which degree each operation is undoable. A positive outcome of this check means a formal guarantee that any sequence of calls to such operations can be undone. A negative outcome contains information on the properties preventing undoability, e.g., which operations are not undoable and why. At runtime we can then warn the user intending to use an irreversible operation; if undo is possible and desired, we apply an AI planning technique to automatically create a workflow that takes the system back to the desired earlier state. We demonstrate the feasibility and applicability of the approach with a prototypical implementation and a number of experiments.

1 Introduction

Cloud computing is now a market larger than US\$ 150 billion worldwide, out of which Infrastructure-as-a-Service (IaaS) is estimated at US\$ 80 billion [8]. Administrators executing applications on an IaaS platform must provision, operate, and release the resources necessary for that application. Administrators perform these activities through the use of a fixed set of API operations - those made available by the IaaS provider. The APIs may be exposed as system administration commands, Web service interfaces, programming language-specific

APIs, or by similar means.

When managing cloud resources, many administrators operate without a safety net – for instance, inadvertently deleting a virtual disk can result in the complete loss of the contained data. Reverting to an earlier state manually is difficult, especially for administrators with limited experience. The facility to rollback a collection of changes, i.e., reverting to a previous acceptable state, a *checkpoint*, is widely recognized as valuable support for dependability [3, 5, 13]. This paper considers the particular needs of administrators managing API-controlled systems to revert to a previous acceptable state. This reversion can be called an “undo” or “rollback”.

The need for undo for administrators is partially motivated by the power of the provided APIs. Such APIs can dramatically improve the efficiency of certain aspects of system administration; but may increase the consequences of human-induced faults. These human induced faults play a large role in overall dependability [27, 28].

To support the administrator managing cloud resources through an API, we aim to provide undoability to a previously known consistent state. If the current state is unsatisfactory for any reason, a consistent earlier state (or a checkpoint) can be restored. This is similar to the concept of “transactional atomicity” in that the period from the consistent earlier state to a future consistent state can be considered as a transaction that may be committed or undone.

However, the nature of a cloud platform introduces particular difficulties in implementing undo. One fundamental issue is the **fixed set of operations**: the administrator cannot alter the set of operations provided by the cloud provider in an API nor even examine its implementation. Administrators have to accept a given API, which is not necessarily designed to support undo. One implication of this inability to change the API is that it is not totally possible to introduce locks to prevent multiple administrators from attempting concurrent undos. It is possible to introduce a lock mechanism on top of our

undo but since administrators can directly use the original API, they could circumvent the locks. We therefore assume that single user undo is enforced by administrative and not technical means.

In the following, we list particular challenges for undoability in the environment outlined above.

1. **Completely irreversible operations.** For some operations, like deleting a virtual disk, no compensating operation is provided. Although it is possible to recover data stored in a disk if the backup exists, a disk itself cannot be restored once it is deleted when no such functionality is supported by a cloud provider.
2. **Partly irreversible operations.** Some operations are mostly undoable, but not fully so. For example, starting a machine seems to be an inverse operation for stopping it. However, certain properties cannot necessarily be restored: any property that is set by the cloud provider when starting a machine cannot be guaranteed to match the value in the checkpointed state. Examples include the startup timestamp, dynamically allocated private or public IP addresses or DNS names, etc. Restoring the exact values of those properties may or may not be important in a given administration task.
3. **As-is API, requiring potentially complex undo workflows.** Since the user can manipulate the state of the resources only through the provided API, restoring a previous acceptable state can only be achieved by *calling the right API operations on the right resources in the right order*. A common assumption is that a sequence of operation calls can be undone by calling inverse operations in the reverse chronological order – e.g., as suggested in ref. [14]; but this is not always correct or optimal [16]. Further, constraints between operation calls can be non-obvious and state-specific. Contrast our accomplishing an undo via provided APIs with undos based on provider furnished tools such as ZFS snapshotting. ZFS snapshotting enables the recovery of a dataset via internal mechanisms. Administrators do not have access to such internal mechanisms and so need to cope with the features the provider made accessible.
4. **Error-prone operations.** Cloud API operations are often themselves error-prone: we frequently observed failures or timeouts on most major commercial cloud platforms. Therefore, failures may occur during the execution of an undo workflow, and need to be handled, such as flexibly executing alternative operations.

To support undoability on existing cloud platforms or similar API-controlled systems, we address the four issues listed above as follows.

1. **Introduction of Pseudo-Delete.** An irreversible operation, like delete, is replaced by corresponding pseudo variant of the operation, like *pseudo-delete* (see e.g., ref. [15]). Pseudo-delete first marks a resource for deletion while retaining ownership; the resource is actually released only when the whole operation sequence is successfully completed.
2. **Undoability Checking.** A major portion of the problem with partly irreversible operations is that their effects are not necessarily known; another portion is that the significance of changes in properties is highly context-specific. We propose a novel method to check the undoability of operations, based on a formal model from AI planning (see e.g., ref. [30]). Given the requirements on which properties need to be restored after the combination of forward and undo workflows, as well as which operations are to be executed, the proposed method determines if these operations can be undone.
3. **Generation of undo workflows.** If rollback is desired our system will automatically generate an undo workflow, i.e., create an appropriate sequence of operation calls from the given API, which, when executed, brings the resources back to their checkpointed state. To this end we use an AI planner [20], based on the formal model of operations mentioned above. Choosing a sequence of operations is a search in the space of possible solutions; highly optimized heuristics solve common cases of this computationally hard problem in reasonable time.
4. **Handling failures on rollback.** We use a particular AI planner [20] that produces plans which can handle failures by including suitable “backup plans”, where such plans exist.

This paper makes the following contributions. We provide undoability support for API-controlled systems, such as cloud management, by checking the undoability of relevant operations. If accomplishing an undo is not always feasible, we can identify the specific operation and specific circumstances under which that operation cannot be undone. If accomplishing an undo is feasible, the undo system from our earlier work¹ can provide an undo workflow when desired. We further developed two prototypes: one is the undoability checker and one is an undo system for Amazon Web Services (AWS)² management operations. Based on these prototypes we evaluated

¹A previous paper [31] was published at the 2012 USENIX HotDep workshop, focused on the undo system. We summarize some of these results here for completeness, but focus on the new aspects: undoability checking, its implications, and new experiments.

²<http://aws.amazon.com>, accessed 30/4/2013

our approach in depth. The website of the work is <http://undo.research.nicta.com.au>.

The remainder of the paper is structured as follows. In Section 2 we describe motivating examples. Section 3 gives an overview of the proposed system. Section 4 discusses the details of the domain model for AWS used in AI planning techniques. Section 5 presents the undoability checking approach, and Section 6 the undo system. In Section 7 we discuss implications for system operations in practice. In Section 8 we evaluate the feasibility of our approach. Section 9 connects and contrasts our work with related research. Section 10 concludes the paper and suggests directions for further study. A technical report contains formal details of the undoability checking omitted here, and is available through the website.

2 Motivating Examples

To showcase the problems addressed in this paper, we provide four concrete system administration scenarios next. These scenarios are mainly taken from our day-to-day system operations using public cloud resources within Yuruware³, a NICTA spin-out offering a backup and disaster recovery solution for cloud-based systems. Some of them are also inspired by products for automating data center operations such as VMware vCenter Orchestrator⁴ and Netflix Asgard⁵. In the evaluation (Section 8.3), we revisit these scenarios and discuss the findings from our evaluation of the specific undoability of each scenario.

Scenario 1. Adding a slave to an existing database server. When traffic to one database server grows, sooner or later it cannot provide enough capacity to handle the workload in time. A common technique to address the issue is to add slave database servers, serving only read queries, while a master server only serves write queries. This reconfiguration is typically performed during scheduled maintenance time, where the database server can be taken offline, and includes two steps: creating slave databases by cloning the database server, and introducing a proxy server to distribute queries among servers.

The first step, i.e., creating slave database servers, consists of the following activities.

1. Stop the instance operating a database server
2. Take a snapshot of the volume storing the database
3. Create copy volumes from the snapshot
4. Launch new instances with a copy volume attached
5. Configure the new instances as slave database servers
6. Start the original instance and configure it as a master

³yuruware.com

⁴vmware.com/products/vcenter-orchestrator

⁵github.com/Netflix/asgard

The second step, i.e., introducing a proxy server, consists of the following activities.

1. Launch a new instance
2. Re-allocate a virtual IP address from a master database to the new instance
3. Configure the new instance as a database proxy

Challenges. If a failure occurs during the reconfiguration, an administrator expects the undo mechanism to remove all newly created resources and restore the state of the instance operating the master database. It is trivial to remove new resources. Properties to be restored are the state (i.e., running), the allocation of the virtual IP address, the instance ID, and the amount of resources allocated. Many properties do not need to be restored such as the public/private IP address allocated to. (In a variant of this scenario, the system may rely on the private IP address of the master database server, such that it must be restorable as well.)

Scenario 2. Scaling up or down an instance. When the underlying platform – such as Amazon Web Services (AWS) – does not support dynamic resource re-allocation (e.g., adding CPU cores without stopping a guest OS), an administrator first needs to stop an instance, change the resource allocation, and start the instance up again.

Challenges. Depending on the context, a variety of properties can be modified on a stopped machine, and need to be restorable. However, stopping a machine on AWS is most likely to change the private IP and the public DNS name.

Scenario 3. Upgrading the Web server layer to a new version. First an administrator creates a new load balancer, then launches instances operating a new version of an application and registers them with the new load balancer. Subsequently, the traffic is forwarded to the new load balancer by updating the respective DNS record. Once all activities completed in the old Web layer, the administrator creates a machine image from one of instances in the old Web layer as a backup and decommissions all resources comprising the old Web layer.

Challenges. In this scenario, all resources that comprise the old Web layer (i.e., the specific instances, a load balancer, their association, etc.) and the DNS record need to be restorable. In a variant of this scenario, instances may be designed to be stateless; hence being able to restore the original number of instances from the same image suffices to undo the changes to the machines.

Scenario 4. Extending the size of a disk volume. Increasing the size of a disk volume in a database server or a file server is a common administration task. Similar to the scenario shown in Scenario 1, an administrator stops an instance, creates a copy of the data volume with

larger size, and swaps the data volume of the instance with the new copy. Then he/she starts the instance again and deletes the old data volume and the snapshot.

Challenges. In this scenario, the original data volume and the association with the instance must be restorable.

3 System Overview

In this section, we summarize the functionality of the undoability checker and the undo system, before giving more details in the next sections. A high-level overview is given in Fig. 1.

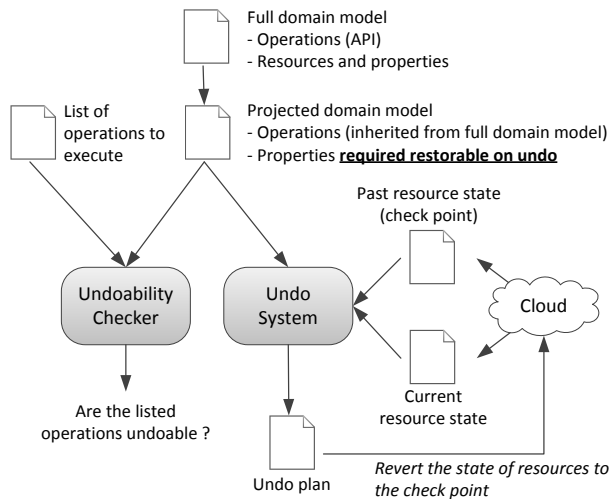


Figure 1: High-level overview of the framework

Our techniques rely on a suitable abstract model of the domain, where each operation has a precise representation of its effects on each aspect of the abstract state, as well as a precise representation of its preconditions, stating in which (abstract) states the operation can be executed. This forms the *full domain model* in Fig. 1, which can, e.g., capture operations from a management API provided by Amazon Web Services (AWS). For instance, the operation deleting a resource such as a disk volume in AWS can only be executed when the disk is available (precondition), and subsequently it is not available but deleted (effect). More details are given in Section 4.

Based on this representation, we extended techniques from AI planning (see e.g., ref. [30] for an overview) to check the undoability of a domain at design time: can each operation be undone in any situation? If no, what are the exact problems? Such problems can be solved by (i) abstracting from details or states that are irrelevant; or (ii) altering the set of operations, e.g., by replacing non-reversible forward operations such as deleting a disk volume through reversible variants, such as *pseudo-delete*. The former can be achieved by altering the domain model. The latter can be achieved by replacing the

delete operation with pseudo-delete in a domain model as well as replacing the actual implementation of delete operation. In Fig. 1 the outcome of the model changes is illustrated as a *projected domain model*. A projected model is used to determine whether a given set of operations (or all operations in a domain) are undoable – whether the state of (projected) resources can be reverted from the state after executing an operation to the state before. Section 5 discusses undoability checking more detail.

When undo is desired, i.e., an administrator wants to revert the state of cloud resources from the current state to a past checkpoint, we use an extended version of the undo system from our previous work [31] to find and execute an undo plan. This is summarized in Section 6. Note that the undo system operates on the projected domain model, so that results from previous undoability checks apply.

4 Domain Model

The domain model formally captures the actions of a domain – in our case, operations defined in a cloud management API. As such, it forms the basis of the undoability checking and the undo system.

While the problem and system architecture are generic, the domain model is of course specific to the management API to be modeled. For our proof-of-concept prototype we focused on Amazon Web Services (AWS) as the domain and chose the planning domain definition language (PDDL) [25] as the planning formalism. We modeled part of the Apache CloudStack API⁶ as well, namely those operations triggering state changes on machines and disk volumes. However, the AWS documentation is more detailed, and the exact effects can be observed from the single implementation of AWS. Therefore, our focus remained on AWS.

One of the most critical aspects for applying AI planning is obtaining a suitable model of the domain [22]. For the purposes of this research, we designed a domain model manually, formulated in PDDL. This model has about 1100 lines of code, and contains representations of 35 actions (see Table 1). Out of these, 18 actions are for managing AWS elastic compute cloud (EC2) resources, such as virtual machines (called *instances*) and disk volumes, and 6 actions for managing the AWS auto-scaling (AS) mechanisms. These actions have been selected due to their high frequency of usage by the developers in our group. Four of the remaining actions are for system maintenance, e.g., switching a server cluster to/from maintenance modes. Those actions are not spe-

⁶cloudstack.apache.org/docs/api/, accessed 21/02/2013.

cific to AWS but defined for generic system administration. The final 7 actions are added undelete actions, one per resource type – which become available by replacing delete with pseudo-delete.

Resource type	API operations
Virtual machine	launch, terminate, start, stop, change VM size, undelete
Disk volume	create, delete, create-from-snapshot, attach, detach, undelete
Disk snapshot	create, delete, undelete
Virtual IP address	allocate, release, associate, disassociate, undelete
Security group	create, delete, undelete
AS group	create, delete, change-sizes, change-launch-config, undelete
AS launch config	create, delete, undelete

Table 1: AWS actions captured in the domain model – adapted from [31]

Case Study: the PDDL definition of the action to delete a disk volume is shown in Listing 1. From this example, it can be seen that parameters are typed, predicates are expressed in prefix notation, and there are certain logical operators (*not*, *and*, *oneof*, ...), also in prefix notation. The precondition requires the volume to be in state *available*, not *deleted*, and not be subject to an *unrecoverable failure*. The effect is either an *unrecoverable failure* or the volume is *deleted* and not *available* any more.

Listing 1: Action to delete a disk volume in PDDL

```

1 (:action Delete-Volume
2  :parameters (?vol - tVolume)
3  :precondition
4    (and
5      (volumeAvailable ?vol)
6      (not (volumeDeleted ?vol))
7      (not (unrecoverableFailure ?vol)))
8  :effect
9    (oneof
10     (and
11       (volumeDeleted ?vol)
12       (not (volumeAvailable ?vol)))
13     (unrecoverableFailure ?vol)))

```

Unrecoverable failure is a predicate we define to model the failure of an action, assuming that the affected resource cannot be brought back to a usable state using API operations. It should be noted that our planning domain model resides on a slightly higher level than the respective APIs. When a planning action is mapped to executable code, pre-defined error handlers are added as well. For example, a certain number of retries and clean-ups take place if necessary. Such a pre-defined error handler, however, works only on the resource in question. If it fails to address an error, an unrecoverable failure is

raised.

From the viewpoint of an AI planner the unrecoverable failure poses two challenges: *non-deterministic actions* and goal reachability. The outcome of *Delete-Volume* (success or unrecoverable failure) is observed as a non-deterministic event. In the presence of non-deterministic actions, the planner has to deal with all possible outcomes, which makes finding a solution harder than in the purely deterministic case. This requires a specific form of planning, called *planning under uncertainty* – see, e.g., Part V in [30].

Further, the question “when is a plan a solution?” arises. To cater for actions with alternative outcomes, a plan may contain multiple branches – potentially including branches from where it is impossible to reach the goal. A branch that contains no unrecoverable failure is the normal case; other branches that still reach the goal are backup branches. A plan that contains more than one branch on which the goal can be reached is called a *contingency plan*. Branches from which the goal cannot be reached indicate situations that require human intervention – e.g., if a specific resource has to be in a specific state to reach the goal, but instead raises an unrecoverable failure, no backup plan is available. This also means the action is not fully undoable (as long as unrecoverable failures are considered).

In planning under uncertainty there are two standard characterizations of plans: a *strong plan* requires *all* branches of a plan to reach the goal, whereas a *weak plan* requires *at least one* branch to reach the goal. Standard planners that can deal with uncertainty are designed to find plans satisfying either of them; however, neither is suitable in our domain. It is highly likely that no strong plan can be found: many of the actions can return an unrecoverable failure, and many of possible branches cannot reach the goal. Weak plans have the disadvantage that only the “happy path” is found: a plan that allows reaching the goal only if nothing goes wrong. When finding a weak plan, a planner does not produce a contingency plan, which we deem insufficient.

In prior work [20], a different notion of a weak plan was introduced: the goal should be reached *whenever it is possible*. This is desired in the setting given here, as it will produce as many branches (i.e., a contingency plan) as possible that still reach the goal, given such alternative branches exist. For finding plans with these solution semantics, a highly efficient standard planner, called *FF* [19], was extended in [20].

There are three discrepancies between the standard AI planning and our use of it in the undo system and the undoability checker, which require attention:

1. In the undo system, when new resources are created after a checkpoint, the resources exist in the initial

state (i.e., the state captured when a rollback is issued) but not in the goal state (i.e., the state when a checkpoint is issued). Unless treated, the AI planner simply leaves these *excess resources* intact: since they are not included in the goal state, they are irrelevant to the planner. However, to undo all changes, excess resources should be deleted. To discover plans that achieve this, we perform a step of preprocessing before the actual planning: the goal state is compared with the initial state in order to find excess resources; the goal state is then amended to explicitly declare that these excess resources should end up in the “deleted” state.

2. In the AI planning variant we employ, new resources cannot simply be created out of nowhere. Instead, we model this case through a unary predicate called *not-yet-created*. In the undoability checker, we consider *not-yet-created* to be equivalent to *deleted*.⁷ Thus, we need to replace any occurrence of *not-yet-created* in an undoability checking goal with *not-yet-created OR deleted*. This allows us to undo effects of actions that create new objects, by simply deleting them.
3. The equals predicate “=” is used in PDDL to mark the equivalence of two constants. In our domain model, we use it in the preconditions of several actions. In the undoability checker, this would cause problems, since the checker derives initial and goal states for planning problems from actions’ preconditions and effects and instantiates them with constants – which can cause contradictions. We circumvent the problem by applying the meaning of the equals predicate in planning tasks created by the undoability checker, and filtering out any contradictory states.

Using these special treatments in the undo system and the undoability checker, we can use a standard PDDL domain model for both purposes – so long as it follows our naming convention for *deleted* and *not-yet-created* predicates.

5 Undoability Checking

As argued in the introduction and the motivating examples, it is not *a priori* clear if all operations can be rolled back under any circumstances. In order to provide the user with confidence, we devised an undoability checker, which uses the domain model described in the previous section. We herein summarize our undoability checking approach – since the problem is highly non-trivial, a separate technical report is available, see Section 1, which provides a full formal treatment of the matter.

⁷This point is related to the above, but different in that the checkpointed state does not contain *not-yet-created* predicates.

5.1 Undoability Checking Overview

Fig. 2 provides an overview of the undoability checker. It involves two separate roles: a tool provider and a user. The tool provider defines the full domain model that formally captures the inputs and the effects of operations available in a cloud platform, such as Amazon Web Services (AWS). The user of the undoability checker is assumed to have enough knowledge to operate systems on a cloud platform; however, he/she does not need to define or know the details of the domain model.

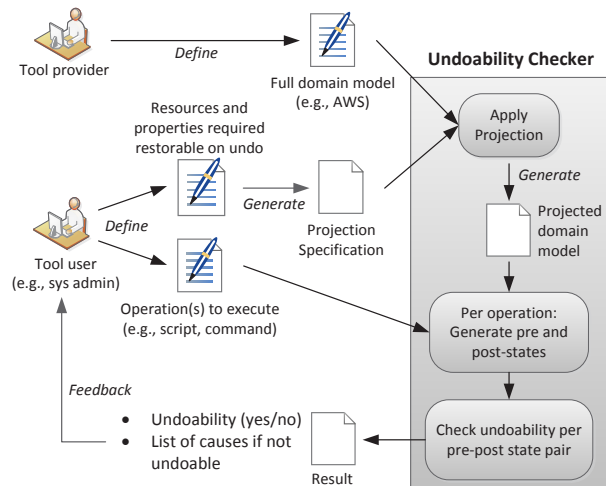


Figure 2: Overview of the undoability checker

The user defines two inputs, both are optional. We refer to these inputs as the *context*. The first one is a list of operations the user plans to execute. It can be a script defining a sequence of operations for a certain system administration task, or a single operation. The other input is a list of resources and properties the user wants to restore upon any failures. It is derived from the context of the system administration as discussed in Section 2. For example, a user may need the IP address of an instance to be restored upon a failure, but not its exact identity.

Given those inputs, the undoability checker examines whether all listed operations are undoable, i.e., the state of all listed resources and properties can be restored after executing these operations. If no operations are provided by the user, the undoability checker tests *all* operations of a domain. If no resources and properties are given by the user, all resources with all their properties from the full domain model are considered. If the output is positive, the user can be sure that the system state can be restored, as long as he/she executes operations listed in the input, or for the whole domain if no operations were specified. If the output is negative, the checker reports the causes – e.g., that a certain property cannot be restored if the user executes a specific operation. The user can consider us-

ing alternative operations or accept a weaker undoability guarantee by giving up the undoability of the property. The tool can of course be used iteratively, until lists of resources, properties, and operations are found which are fully undoable.

5.2 Domain Model Projection

Undoability of the full domain indicates that all types of resources and the properties can be restored after executing any available operations in a cloud platform. Depending on system administration tasks, the part of the domain that is required to be undoable might be significantly smaller than the full domain model, however. Therefore, before the actual undoability check is performed, the undoability checker considers the context provided by the user, if any, and extracts a subset of the full domain. To achieve this, we introduce the concept of a *projection of a domain model*.

From the list of resources and properties in the context, the undoability checker creates a projected domain that only captures resources and their properties involved in a provided context. This is done by creating a projection specification in a specific format from the user input, and applying the specification to the full domain. If no list of relevant resources and properties is provided by the user, the full domain is used instead of a projected one. Projections can also include a role-player concept, such that e.g., virtual machines can be said to play a role (e.g., Web server), where their actual identity is less relevant. A projected domain model is itself a valid domain model in PDDL, so undoability checking can proceed as per below, with or without projection.

In terms of our running example of AWS, without projection most actions in the AWS domain model would not be considered undoable, due to the unrecoverable errors discussed in Section 4. If an unrecoverable error occurs, one cannot reverse back to the previous state of that resource. Unless using a different resource to fulfill the same purpose (e.g., a new machine instead of a crashed one, from the same machine image) is acceptable, the unrecoverable error is just that: unrecoverable. More interesting is the question: *assuming no unrecoverable error occurs, do we have guaranteed undoability?* This question can be addressed by leveraging domain projection: the user can define a context that excludes unrecoverable errors.

Even so, most real-world domains do not require that each and every aspect can be restored. For example, a virtual machine in AWS has over 20 attributes such as the resource identity assigned by the provider (e.g., ID `i-d21fa486`), public and private DNS names assigned by the provider (e.g., `ec2-xxx.compute.amazonaws.com`), instance type indicating the amount of computing re-

sources allocated (e.g., `m1.small` roughly equals to 1.2GHz CPU and 1.7GB RAM), the identity of the machine image used to launch the virtual machine (e.g., `ami-a6a7e7f4`) and launch timestamp. Although some of those properties are easy to be restored, others are not. For example, to obtain an instance whose machine image identity is the same as that of a terminated one, an administrator simply launches a new instance from the machine image used before. However, there is no way to obtain the same resource identity or public DNS name once an instance is terminated, since these are assigned dynamically by the cloud provider.

5.3 Undoability Checking Algorithm

From a (projected) domain, the checker considers each relevant action individually (either from the list provided by the user, or all actions in the domain). For each relevant action, the checker tries to find situations in which executing the action *may* result in a state from which the system *cannot* get back to the state before the action was executed. If no such situation exists, we call the action *fully undoable*. If each action can be reversed individually, then any sequence of actions can be reversed – this is related to the Sagas approach [14], which is discussed in Section 9. While such a sequence may be highly suboptimal, we only use it to prove undoability on a theoretical level – when the actual situation occurs, our undo system finds a shorter undo workflow.

To check undoability of an action, we need to ascertain undoability for any state in which the action could be executed – the *pre-states* of the action. In general, this is an infinite set. We derive a *sufficient* set of pre-states for undoability checking – i.e., a set that captures any distinct situation from the pre-states – by considering:

1. how many constants of which type may be used by any of the actions in the domain;
2. analyzing which actions may contribute to undoing the one currently being checked;
3. deriving any combination of the logical statements from the combination of the above sets.

Based on the sufficient set of pre-states, we compute the set of possible post-states – the outcomes of applying the action being checked to each of the pre-states, where non-determinism means a pre-state can result in multiple post-states. For each post-state, we then check if the corresponding pre-state is reachable, by formulating a respective planning task and feeding it into the AI planner [20]. The usual *planning problem* [30] is the following: given formal descriptions of the *initial state* of the world, the desired *goal state*, and a set of available *actions*, find a sequence of actions that leads from the initial to the

goal state. In the case of the undoability checker, the question is: can the pre-state be reached from the post-state? Thus, the post-state is the initial state of the planning problem, and the corresponding pre-state forms the goal state. If there is a plan for each pre-post-state pair, then full undoability (in the given context) is shown; else, the problematic cases are reported back to the tool user.

5.4 Checker Usage Models

There are two ways to deploy the undoability checking: offline and online. The offline deployment model is used for checking a whole domain, or a specific script / workflow implementing a system administration task. When used by administrators, this deployment model assumes that the script / workflow will be executed in future, e.g., during scheduled downtime on the following weekend, and the tool user prepares for it. In this scenario administrators usually spend enough time on developing a plan in order to get maximum leverage out of the scheduled maintenance period. Given a list of operations invoked in a script⁸, the undoability checker examines if they are reversible. The checker shows a list of operations not reversible if exists (Figure 2.) The user is expected to use the undoability checker iteratively to improve the quality of scripts to execute. For example, removing all irreversible operations from the script, placing irreversible operations as late as possible to make the workflow reversible until calling one of the irreversible operations, or altering the set of attributes to be undoable such as using a virtual IP address instead of an internal IP address.

The alternative deployment model, i.e., online deployment model, is used for checking the undoability of each operation an administrator is about to execute. This deployment model provides a *safety net* to an administrator directly executing operations on a console. In this scenario, each operation is executed through a wrapper (or a proxy) rather than executed directly on cloud resources, as discussed in the next section. Before an operation is executed on cloud resources, the wrapper performs an undoability check and sends a warning to the administrator if the operation is not undoable.

The two models are built on different assumptions. The offline model assumes that the concrete state of resources such as IP addresses cannot be truthfully observed, since the analysis is performed before a script is actually executed. Therefore, it determines the undoability of all operations against *all possible* pre-states. This mode often results in very strong undoability guarantees – arguably stronger than needed, since the undoability checker examines pre-states that may never occur in the

actual system. Say, for example, a machine had a fictitious attribute 'unmodified since start', which would be true initially after starting a machine, but false after the first change was applied. In general, any action modifying the machine, e.g., changing its size, would be undoable – and detected as such by the offline check. However, the online check might encounter a state where this attribute was already false – in this particular case, modifications to the machine would be undoable.

To address the limitation of the offline model, the online model assumes that it can obtain the status of resources by making calls to the API a cloud platform provides. If an operation is not known to be fully undoable in a given administration context (by looking up the result that the offline check provides), it senses the state of the resources and forwards this information to the undoability checker. The checker then takes this state as the only pre-state, and checks if all possible outcomes of executing the operation in this *specific* pre-state are undoable. Therefore, operations identified as not undoable by the offline check could be identified as undoable by the online check depending on the status of resources.

Although the online model is very targeted in terms of the performed undoability checks, it may not be practical depending on the responsiveness of APIs. It is not uncommon for a scan of the resources in public cloud platform to take longer than 30 seconds depending on the network latency, while it can be less than 1 second on an on-premise virtualized environment. Depending on the user's preferences, the slow responsiveness may be unacceptable.

6 Undo System Design

The main component of the undo system we propose concerns automatically finding a sequence of operations for realizing rollback. An earlier version of the undo system was described in [31], which we summarize here.

6.1 Overview of the Undo System

Fig. 3 shows the overview of the undo system, which is partially positioned between the user / operational script and the cloud management API. An administrator or an operational script first triggers a *checkpoint* to be taken⁹. Our undo system gathers relevant information, i.e., state of cloud resources and their relationship, at that time. After a checkpoint, the system starts to offer rollback to the checkpoint or committing the changes. Before either of these command is called, the system transparently replaces certain non-reversible operations, e.g., deleting a

⁸Our current implementation does not parse a script, e.g., a bash script, directly to extract operations. We assume the user provides a list of operations used in a script to the checker.

⁹While the resources can form a vastly distributed system, their state information is obtained from a single source of truth: AWS's API.

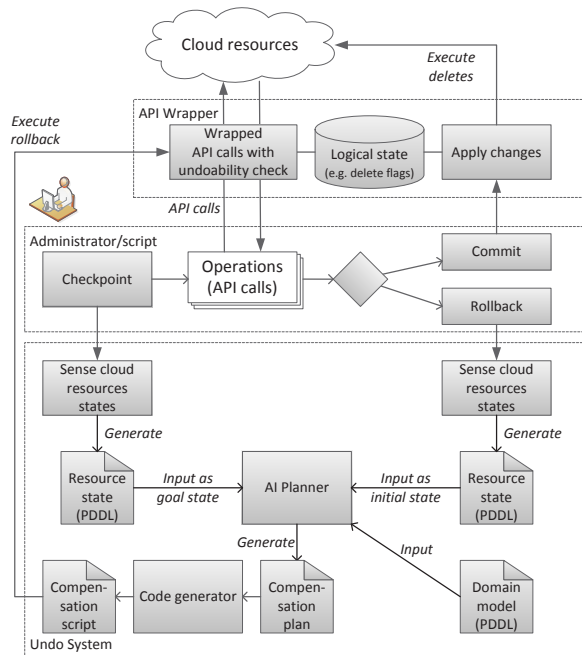


Figure 3: Overview of rollback via planning – adapted from [31]

resource, with reversible ones, like pseudo-delete. Further, it checks the undoability of each operation to be executed, as discussed in Section 5.4. When a *commit* is issued, the non-reversible changes are applied to the cloud resources – thus, rollback is not offered anymore. When a *rollback* is issued, the system again gathers the state of cloud resources and feeds the pair of state information to an AI planner to construct an undo sequence of operations, as discussed in Section 6.3.

6.2 API Wrapper

The undo system offers a wrapper for cloud APIs. After a checkpoint, when the user asks to delete a resource, the wrapper sets a *delete flag* (or *ghost flag*), indicating that the resource is logically deleted. Note that this requires altering *each* subsequent API call to the resource through the wrapper: queries to list provisioned resources need to be altered in their responses by filtering flagged resources out; changes to flagged resources need to be caught and answered, e.g., by returning a “not found” error message.

When a delete operation is to be reversed – triggered by a respective *undelete* operation – the wrapper simply removes the delete flag. When the user requests a rollback, the AI planner includes an undelete call on that resource. Only when a commit is issued, all resources with a delete flag are physically deleted.

The wrapper is also the connection point to the un-

doability checker in the online deployment model (Section 5.4). If an action is not undoable, it warns the user, with an option to cancel.

6.3 Rollback via AI Planning

For rollback, the goal is to return to the state of the system when a checkpoint was issued. Our undo system finds a sequence of undo actions by using the AI planner from [20]. In undo planning, the initial state is the state of the system when rollback was issued. The desired goal state is the system state captured in a checkpoint. The available actions are the API operations offered by a cloud provider, captured in the domain model.

In the undo system (Fig. 3), the planning problem (initial and goal states, set of available actions) is the input to the planner. Its output is a workflow in an abstract notation, stating which action to call, in which order, and for which resources. The abstract workflow is forwarded to a code generator, which transforms it into executable code. The final step is to execute the workflow against the wrapped API operations.

7 Implications for Practice

In this section, we discuss how the various parts of the systems can be used by practitioners in the future. The easiest case is when a domain is fully undoable. The implication is that the user can do anything, and have the confidence that the undo system can find a path back to any checkpoint. Similarly, if a script consists only of fully undoable actions, employing it can be done with high confidence.

If properties need to be excluded through projection to achieve undoability, the administrator gains explicit knowledge of the boundaries of undoability. This knowledge can be used for improving the dependability of administration tasks through awareness, or to inform changes to scripts or procedures. If no alternative action exists, a script / workflow can potentially be *re-ordered*, to schedule non-undoable actions as late as possible. The rationale for doing so is that executing an irreversible action can be seen as a form of commit: if anything goes wrong before, it can still be undone; afterwards this is not the case. As many reversible operations as possible should therefore be done before the irreversible one(s).

Another way to leverage the results is to provide a *safe mode* of an API, which includes the subset of operations that are fully undoable. This subset can e.g., be used in day-to-day operations, while using reversible operations might be reserved for users with higher privileges. Untrusted or less experienced administrators may only be allowed to operate within the boundaries of the safe

mode, so that any mistakes or malicious behavior can always be undone.

Finally, the undo system's checkpointing / commit / rollback commands can be made part of scripts or workflows. For instance, a script may always set a checkpoint before execution. If any of a set of known errors arises, rollback is automatically triggered. If a final set of automated checks succeeds, the changes can be committed before exiting the script.

8 Implementation and Evaluation

In this section, we describe our prototypical implementations and the experiments we conducted to evaluate our approach. The latter include applying the undoability checker to the AWS domain model, assessing the undoability of the scenarios from Section 2, as well as a summary of performance experiments from our earlier work [31].

8.1 Implementation

Both the undo system and the undoability checker have been implemented as prototypes. The undo system has been rolled out for internal beta-testing and used for feasibility and performance experiments. Both prototypes have been implemented in Java, making command line calls to the version of the FF planner used in our work [20]. The undo system further includes bash scripts for use as a command line tool, replacing the AWS implementations of the operations in Table 1 and providing additional operations for checkpointing, undelete, rollback, and commit.

The undoability checker has around 17,500 lines of Java code, including an extended version of an open source parser, lexer, and PDDL data model¹⁰. The undo system has roughly 7,300 lines of Java code, the FF planner as per above around 17,000 lines of code in C and lexer / parser definitions.

The main limitations of the current prototypes are the following.

- The undoability checker does not have a component for generating projections as yet – desired projections had to be applied by manually creating variants of the Amazon Web Services (AWS) domain model.
- The API wrapper of the undo system is not integrated with the undoability checker, although the latter has an interface that can be adapted to implement this integration (checking undoability of single actions in specific states).

¹⁰<http://www.zeynsaigol.com/software/graphplanner.html>, accessed 8/4/2013

- Neither tool has been optimized for performance yet, and both are still in the state of (well-tested) prototypes. Particularly for the undo system, we plan to change this in the near-term future.

8.2 Undoability of a Full Domain

Using the implementation of the undoability checker, we performed several iterations of checking the AWS domain model. As expected, the domain model including *unrecoverable errors* is not undoable. After implementing a projection to remove *unrecoverable errors* as well as several changing parameters such as internal IP address, undoability is given for most actions. Note, that this domain model includes our AWS extension of undelete actions.

In comparison to the earlier version of our domain model, as described in [31], it should be noted that we added more details in all but a few actions. This primarily concerns statements in preconditions and effects that were previously seen as not required. For instance, deleting a volume (see Listing 1) now has a precondition of *not volume deleted*, which was previously seen as implicitly following from *volume available*. We further split up certain actions, without implications for practice – e.g., instead of changing all auto-scaling group targets (min, max, desired) at once, for the purposes of undoability checking we split this action into three separate ones. This reduces the number of separate planning tasks significantly. Due to such changes, the domain model grew from 800 to 1100 lines of code.

Our current results of checking undoability for the whole domain model (Table 1) show 34 out of the 35 actions to be fully undoable, given the above projection. For this purpose, the prototype ran for 11s inside a VirtualBox VM on two (hyperthreaded) cores from an Intel i7-2600 quadcore CPU @ 3.4GHz, with 4GB RAM available to it. During that time, it called the planner with 1330 separate planning problems – i.e., pre-post-state pairs as per Section 5.3.

The action for which undoability cannot currently be shown is *create auto-scaling group*. This action creates an AS group with the respective three sizes (min, max, desired). The corresponding *delete* action has only the precondition of *max = 0* (which in practice implies *min = 0 = desired*). However, this implication is not expressed to the planner as yet. If it were, then the number of possible instantiations in pre/post-state pairs becomes too large to be handled by the current implementation. We believe that this issue can be solved in the future, and undoability can be fully shown, when applying the above-mentioned projection.

8.3 Undoability of Administration Tasks

We next present our evaluation of the undoability of concrete system administration scenarios. As described in Section 5.4, we assume that the tool user provides a list of operations to execute in system administration task. The undoability checker examines whether the list contains irreversible operations given a set of properties to be reversible. This section discusses the results that the undoability checker produces for the four scenarios introduced in Section 2.

Findings for Scenario 1. Adding a slave to an existing database server. All steps can all be undone through the AWS management API. In particular, the original configuration of the database server can be restored by restoring the volume from a snapshot taken – however, while this step is on top of our future work list, it is currently not implemented in our undo system. The variant relying on the private IP address of the database server can be undone, so long as *stop machine* is replaced with *pseudo-stop*, and assuming that reconfiguring the database server can be done on a running machine. Note that replacing *stop* with *pseudo-stop* has the side-effect that no changes such as changing the machine size can be done: such changes require the machine to be actually stopped, in which case the private IP cannot be restored.

Findings for Scenario 2 Scaling up or down an instance. In our current model, we support changing the instance size. Other aspects could be modeled analogously: from studying the AWS documentation of `ec2-modify-instance-attribute`¹¹, we believe any aspect that can be changed can also be undone. As for the implicit changes to private IP / public DNS name, the undo system could be configured to warn the user.

Findings for Scenario 3. Upgrading the Web server layer to a new version. DNS records are outside the scope of our domain, and need to be undone manually. Load balancers and the creation/deletion of machine images are currently not captured, but could be in the future. The remaining actions are all shown to be undoable in our domain model.

Findings for Scenario 4. Extending the size of a disk volume. This scenario uses solely operations shown to be undoable. Since the machine is stopped by the administrator at any rate, its private IP will change, so that undoability is not affected. The warnings mentioned in Scenario 3 can be applied here as well.

¹¹<http://docs.aws.amazon.com/AWSEC2/latest/CommandLineReference/ApiReference-cmd-ModifyInstanceAttribute.html>, last accessed 29/4/2013

8.4 Undo System Performance

We summarize the performance results from [31], and discuss performance changes resulting from the changes to the domain explained in Section 8.2. For this experiment, we assembled over 70 undo planning tasks. When tasked with finding the solution to undo problems requiring up to 20 undo actions, the execution time of the planner was often close to or below the resolution of measurement (10 ms) – i.e., all such plans were found in less than 0.01 seconds. Even plans with more than 60 actions were found in less than 2 seconds, using the domain model from [31]. However, after altering the domain model, planning times changed as follows. Plans with less than 30 steps were still found in less than 10ms. More complex plans took 8-15 times longer than with the original domain model. To put this into perspective: scripts with over 20 steps are unusual in our groups’ experience. In comparison to the execution time of scripts on AWS – for instance, the average execution time over 10 runs of an 8-step undo plan of AWS operations was 145 seconds – the cost for planning is marginal in many situations.

9 Related Work

Our undo approach is a checkpoint-based rollback method [13, 29]. Alternative rollback methods are log-based [13, 15] or using “shadow pages” in databases [15]. In much research, checkpoints store a relevant part of the state on memory and/or disk, and for rollback it suffices to copy the saved information back into. In contrast, rollback in our setting means achieving that the “physical state” of a set of virtual resources matches the state stored in a checkpoint – i.e., achieving rollback requires executing operations, not only copying information.

Thus, our setting is similar to long-running transactions in workflows or business processes. [16] gives an overview of approaches for failure and cancellation mechanisms for long-running transactions in the context of business process modeling languages, where typical mechanisms are flavors or combinations out of the following: (i) Sagas-style reverse-order execution of compensators [14]; (ii) short-running, ACID-style transactions; and (iii) exception handling similar to programming languages like C++ and Java. As for (i), on cloud platforms, compensating operations may not be available. Even when an operation is an inverse for another, there may be non-obvious constraints and side-effects, so that executing the apparently compensating operations in reverse chronological order would not restore the previous system state properly; a different order, or even different operations, might be more suitable [16]. This argument supports our approach to use an AI planner for

coming up with a targeted undo plan. ACID cannot be achieved so (ii) is unsuitable in our setting: the state of the cloud resources is externalized instantaneously, so consistency and isolation are impossible to retain here. As for (iii), hand-coded exception handling for whole workflows can be implemented, but is error-prone; per-operation exception handling is unsuitable, since the behavior of an action is non-deterministic, context-specific, and the target state is not static. In summary, the traditional approaches are not a good fit for the problem at hand.

Besides AI Planning, other techniques were used to achieve dependability in distributed systems management. [21] uses POMDPs (partially-observable Markov decision processes) for autonomic fault recovery. An architecture-based approach to self-repair is presented in [4]. [26] is a self-healing system using case-based reasoning and learning. The focus in these works is self-repair or self-adaptation, not undo for rollback.

AI Planning has been used several times for system configuration, e.g., [1, 2, 7, 9, 10, 11, 23], and for cloud configuration, e.g., [17, 18, 24]. Some works aim at reaching a user-declared goal, e.g., [1, 7, 17, 18, 23], whereas others target failure recovery, e.g., [2, 9, 10, 24]. The goal in the latter case is to bring the system back into an operational state after some failure occurred. The closest related work are [17, 18, 24]. In [18], planning is applied in a straight-forward fashion to the problem of reaching a user-specified goal. The work is well integrated with cloud management tools, using Facter, Puppet and ControlTier – all of which experience some level of popularity among administrators nowadays. [24] applies hierarchical task network (HTN) planning on the PaaS level for fault recovery. [17] uses HTN planning to achieve configuration changes for systems modeled in the common information model (CIM) standard.

As for the undoability checking, Burgess and Couch [6] have shown that well-defined rollback is impossible to achieve in realistic systems, mostly due to non-determinism and openness. This result concurs with our argumentation for the need of projection: only fully closed and deterministic systems have a chance of being fully undoable without projection. For any other case, our approach first can point out what prevents undoability, and iteratively those issues can be projected away, so as to understand under which assumptions undoability is given. Our undo tool takes non-determinism of actions into account directly.

Our undoability checking extends a model of reversible actions from Eiter et al. [12]. Our work differs from [12] in two major ways: we consider undoability of full and projected domains, and we assume the state to revert to is known. This changes the underlying formalism (described in the technical report – see Section 1),

and hence most formal results built on top of it.

10 Conclusions

In this paper we describe our support for undoability, building on two approaches: (i) an undoability checker analyses to what degree operations can be rolled back; and (ii) an undo system that automatically generates rollback workflows, when the need arises. The latter can essentially provide transactional atomicity over API-controlled environments, like cloud management. We evaluated the approaches through the prototypes we developed, with performance experiments and by applying them to real-world examples of cloud management APIs and best practices. An intrinsic limitation of our approaches is that they operate on a manually created model of the available operations. While we took care to assess that the model truthfully captures the implementation, this cannot be formally guaranteed without access to AWS's API implementation, deployment, and operation of the cloud platform. Further, the model is not going to be aware of future changes to the API and its implementation.

The prototype for the undo system is in the process of being developed into a more mature tool, and we may make it available for public use through our website – see Section 1. It will likely feature different levels of undoability from which the users can choose, as established through the undoability checker.

In future work, we plan to extend the undoability checker with an approach to find projections leading to full undoability automatically, such that the removed properties are minimal. For the undo system, we plan an extension to capture the internal state of resources when checkpointing and to restore the internal state on rollback. For example, the content of a disk volume can be captured by taking a snapshot. Finally, the undo system will be extended to handle multiple checkpoints and manage them by their names, where administrators can then choose to rollback to checkpoint P_1 or commit all changes up to checkpoint P_2 .

Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. This work is partially supported by the research grant provided by Amazon Web Services in Education ¹². We thank Gernot Heiser for his helpful remarks on this paper.

¹²<http://aws.amazon.com/grants/>

References

- [1] ARSHAD, N., HEIMBIGNER, D., AND WOLF, A. L. Deployment and dynamic reconfiguration planning for distributed software systems. In *ICTAI'03: Proc. of the 15th IEEE Intl Conf. on Tools with Artificial Intelligence* (2003), IEEE Press, p. 3946.
- [2] ARSHAD, N., HEIMBIGNER, D., AND WOLF, A. L. A planning based approach to failure recovery in distributed systems. In *WOSS'04: 1st ACM SIGSOFT Workshop on Self-Managed Systems* (2004), pp. 8–12.
- [3] AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B., AND LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 11–33.
- [4] BOYER, F., DE PALMA, N., GRUBER, O., SICARD, S., AND STEFANI, J.-B. A self-repair architecture for cluster systems. In *Architecting Dependable Systems VI*, vol. 5835 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009, pp. 124–147.
- [5] BROWN, A., AND PATTERSON, D. Rewind, repair, replay: three R's to dependability. In *10th ACM SIGOPS European workshop* (2002), pp. 70–77.
- [6] BURGESS, M., AND COUCH, A. L. On system rollback and totalized fields: An algebraic approach to system change. *Journal of Logic and Algebraic Programming* 80, 8 (2011), 427–443.
- [7] COLES, A. J., COLES, A. I., AND GILMORE, S. Configuring service-oriented systems using PEPA and AI planning. In *Proceedings of the 8th Workshop on Process Algebra and Stochastically Timed Activities (PASTA 2009)* (August 2009).
- [8] DA ROLD, C., JESTER, R., MAURER, W., CHAMBERLIN, T., ENG, T. C., AND PETRI, G. Data center services: Regional differences in the move toward the cloud. Tech. Rep. G00226699, Gartner Research, 29 February 2012.
- [9] DA SILVA, C. E., AND DE LEMOS, R. A framework for automatic generation of processes for self-adaptive software systems. *Informatica* 35 (2011), 3–13. Publisher: Slovenian Society Informatika.
- [10] DALPIAZ, F., GIORGINI, P., AND MYLOPOULOS, J. Adaptive socio-technical systems: a requirements-driven approach. *Requirements Engineering* (2012). Springer, to appear.
- [11] DRABBLE, B., DALTON, J., AND TATE, A. Repairing plans on-the-fly. In *Proc. of the NASA Workshop on Planning and Scheduling for Space* (1997).
- [12] EITER, T., ERDEM, E., AND FABER, W. Undoing the effects of action sequences. *Journal of Applied Logic* 6, 3 (2008), 380–415.
- [13] ELNOZAHY, E. N. M., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.* 34, 3 (Sept. 2002), 375–408.
- [14] GARCIA-MOLINA, H., AND SALEM, K. Sagas. In *SIGMOD'87: Proc. Intl. Conf. on Management Of Data* (1987), ACM, pp. 249–259.
- [15] GRAEFE, G. A survey of b-tree logging and recovery techniques. *ACM Trans. Database Syst.* 37, 1 (Mar. 2012), 1:1–1:35.
- [16] GREENFIELD, P., FEKETE, A., JANG, J., AND KUO, D. Compensation is not enough. *Enterprise Distributed Object Computing Conference, IEEE International 0* (2003), 232.
- [17] HAGEN, S., AND KEMPER, A. Model-based planning for state-related changes to infrastructure and software as a service instances in large data centers. In *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing* (Washington, DC, USA, 2010), CLOUD '10, IEEE Computer Society, pp. 11–18.
- [18] HERRY, H., ANDERSON, P., AND WICKLER, G. Automated planning for configuration changes. In *LISA'11: Large Installation System Administration Conference* (2011).
- [19] HOFFMANN, J., AND NEBEL, B. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)* 14 (2001), 253–302.
- [20] HOFFMANN, J., WEBER, I., AND KRAFT, F. M. SAP speaks PDDL: Exploiting a software-engineering model for planning in business process management. *Journal of Artificial Intelligence Research (JAIR)* 44 (2012), 587–632.
- [21] JOSHI, K. R., SANDERS, W. H., HILTUNEN, M. A., AND SCHLICHTING, R. D. Automatic model-driven recovery in distributed systems. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems* (Washington, DC, USA, 2005), SRDS '05, IEEE Computer Society, pp. 25–38.
- [22] KAMBHAMPATI, S. Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain models. In *AAAI'07: 22nd Conference on Artificial Intelligence* (2007).
- [23] LEVANTI, K., AND RANGANATHAN, A. Planning-based configuration and management of distributed systems. In *Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management* (Piscataway, NJ, USA, 2009), IM'09, IEEE Press, pp. 65–72.
- [24] LIU, F., DANCIU, V., AND KERESTEY, P. A framework for automated fault recovery planning in large-scale virtualized infrastructures. In *MACE 2010, LNCS 6473* (2010), pp. 113–123.
- [25] MCDERMOTT, D., ET AL. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Committee, 1998.
- [26] MONTANI, S., AND ANGLANO, C. Achieving self-healing in service delivery software systems by means of case-based reasoning. *Applied Intelligence* 28 (2008), 139–152.
- [27] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4* (Berkeley, CA, USA, 2003), USITS'03, USENIX Association, pp. 1–1.
- [28] PATTERSON, D., AND BROWN, A. Embracing failure: A case for recovery-oriented computing (ROC). In *HPTS'01: High Performance Transaction Processing Symposium* (2001).
- [29] RANDELL, B. System structure for software fault tolerance. *IEEE Transactions On Software Engineering* 1, 2 (1975).
- [30] TRAVERSO, P., GHALLAB, M., AND NAU, D., Eds. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2005.
- [31] WEBER, I., WADA, H., FEKETE, A., LIU, A., AND BASS, L. Automatic undo for cloud management via AI planning. In *Hot-Dep'12: Proceedings of the Workshop on Hot Topics in System Dependability* (Oct. 2012).

Back to the Future: Fault-tolerant Live Update with Time-traveling State Transfer

Cristiano Giuffrida Călin Iorgulescu Anton Kuijsten Andrew S. Tanenbaum

Vrije Universiteit, Amsterdam

{giuffrida, calin.iorgulescu, akuijst, ast}@cs.vu.nl

Abstract

Live update is a promising solution to bridge the need to frequently update a software system with the pressing demand for high availability in mission-critical environments. While many research solutions have been proposed over the years, systems that allow software to be updated on the fly are still far from reaching widespread adoption in the system administration community. We believe this trend is largely motivated by the lack of tools to automate and validate the live update process. A major obstacle, in particular, is represented by state transfer, which existing live update tools largely delegate to the programmer despite the great effort involved.

This paper presents *time-traveling state transfer*, a new automated and fault-tolerant live update technique. Our approach isolates different program versions into independent processes and uses a semantics-preserving state transfer transaction—across multiple *past*, *future*, and *reversed* versions—to validate the program state of the updated version. To automate the process, we complement our live update technique with a generic state transfer framework explicitly designed to minimize the overall programming effort. Our time-traveling technique can seamlessly integrate with existing live update tools and automatically recover from arbitrary run-time and memory errors in any part of the state transfer code, regardless of the particular implementation used. Our evaluation confirms that our update techniques can withstand arbitrary failures within our fault model, at the cost of only modest performance and memory overhead.

1 Introduction

In the era of pervasive and cloud computing, we are witnessing a major paradigm shift in the way software is developed and released. The growing demand for new features, performance enhancements, and security fixes translates to more and more frequent software up-

dates made available to the end users. In less than a decade, we quickly transitioned from Microsoft’s “*Patch Tuesday*” [39] to Google’s “*perpetual beta*” development model [67] and Facebook’s tight release cycle [61], with an update interval ranging from days to a few hours.

With more frequent software updates, the standard halt-update-restart cycle is irremediably coming to an impasse with our growing reliance on nonstop software operations. To reduce downtime, system administrators often rely on “*rolling upgrades*” [29], which typically update one node at a time in heavily replicated software systems. While in widespread use, rolling upgrades have a number of important shortcomings: (i) they require redundant hardware, which may not be available in particular environments (e.g., small businesses); (ii) they cannot normally preserve program state across versions, limiting their applicability to stateless systems or systems that can tolerate state loss; (iii) in heavily replicated software systems, they lead to significant update latency and high exposure to “*mixed-version races*” [30] that can cause insidious update failures. A real-world example of the latter has been reported as “*one of the biggest computer errors in banking history*”, with a single-line software update mistakenly deducting about \$15 million from over 100,000 customers’ accounts [43].

Live update—the ability to update software on the fly while it is running with no service interruption—is a promising solution to the update-without-downtime problem which does not suffer from the limitations of rolling upgrades. A key challenge with this approach is to build trustworthy update systems which come as close to the usability and reliability of regular updates as possible. A significant gap is unlikely to encourage adoption, given that experience shows that administrators are often reluctant to install even regular software updates [69].

Surprisingly, there has been limited focus on automating and validating generic live updates in the literature. For instance, traditional live update tools for C programs seek to automate only basic type transforma-

tions [62, 64], while more recent solutions [48] make little effort to spare the programmer from complex tasks like *pointer transfer* (§5). Existing live update validation tools [45–47], in turn, are only suitable for *offline* testing, add no *fault-tolerant* capabilities to the update process, require *manual* effort, and are inherently *update timing-centric*. The typical strategy is to verify that a given test suite completes correctly—according to some manually selected [45, 46] or provided [47] specification—regardless of the particular time when the update is applied. This testing method stems from the extensive focus on live update timing in the literature [44].

Much less effort has been dedicated to automating and validating *state transfer* (ST), that is, initializing the state of a new version from the old one (§2). This is somewhat surprising, given that ST has been repeatedly recognized as a challenging and error-prone task by many researchers [13, 22, 23, 57] and still represents a major obstacle to the widespread adoption of live update systems. This is also confirmed by the commercial success of Ksplice [11]—already deployed on over 100,000 production servers [4]—explicitly tailored to small security patches that hardly require any state changes at all (§2).

In this paper, we present *time-traveling state transfer* (TTST), a new live update technique to automate and validate generic live updates. Unlike prior live update testing tools, our validation strategy is *automated* (manual effort is never strictly required), *fault-tolerant* (detects and immediately recovers from any faults in our fault model with no service disruption), *state-centric* (validates the ST code and the full integrity of the final state), and *blackbox* (ignores ST internals and seamlessly integrates with existing live update tools). Further, unlike prior solutions, our fault-tolerant strategy can be used for *online* live update validation in the field, which is crucial to automatically recover from unforeseen update failures often originating from differences between the testing and the deployment environment [25]. Unlike commercial tools like Ksplice [11], our techniques can also handle complex updates, where the new version has significantly different code and data than the old one.

To address these challenges, our live update techniques use two key ideas. First, we confine different program versions into independent processes and perform *process-level* live update [35]. This strategy simplifies state management and allows for automated state reasoning and validation. Note that this is in stark contrast with traditional *in-place* live update strategies proposed in the literature [10–12, 22, 23, 58, 62, 64], which “glue” changes directly into the running version, thus mixing code and data from different versions in memory. This mixed execution environment complicates debugging and testing, other than introducing address space fragmentation (and thus run-time performance overhead) over time [35].

Second, we allow two process-level ST runs using the time-traveling idea. With time travel, we refer to the ability to navigate backward and forward across program state versions using ST. In particular, we first allow a *forward* ST run to initialize the state of the new version from the old one. This is already sufficient to implement live update. Next, we allow a second *backward* run which implements the reverse state transformation from the new version back to a copy of the old version. This is done to validate—and safely rollback when necessary—the ST process, in particular to detect specific classes of programming errors (i.e., memory errors) which would otherwise leave the new version in a corrupted state. To this end, we compare the program state of the original version against the final state produced by our overall transformation. Since the latter is semantics-preserving by construction, we expect differences in the two states *only* in presence of memory errors caused by the ST code.

Our contribution is threefold. First, we analyze the state transfer problem (§2) and introduce *time-traveling state transfer* (§3, §4), an automated and fault-tolerant live update technique suitable for online (or offline) validation. Our TTST strategy can be easily integrated into existing live update tools described in the literature, allowing system administrators to seamlessly transition to our techniques with no extra effort. We present a TTST implementation for user-space C programs, but the principles outlined here are also applicable to operating systems, with the process abstraction implemented using lightweight protection domains [72], software-isolated processes [53], or hardware-isolated processes and microkernels [50, 52]. Second, we complement our technique with a TTST-enabled state transfer framework (§5), explicitly designed to allow arbitrary state transformations and high validation surface with minimal programming effort. Third, we have implemented and evaluated the resulting solution (§6), conducting fault injection experiments to assess the fault tolerance of TTST.

2 The State Transfer Problem

The state transfer problem, rigorously defined by Gupta for the first time [41], finds two main formulations in the literature. The traditional formulation refers to the live initialization of the data structures of the new version from those of the old version, potentially operating structural or semantic data transformations on the fly [13]. Another formulation also considers the execution state, with the additional concern of remapping the call stack and the instruction pointer [40, 57]. We here adopt the former definition and decouple *state transfer* (ST) from *control-flow transfer* (CFT), solely concerned with the execution state and subordinate to the particular update mechanisms adopted by the live update tool

```

--- a/drivers/md/dm-crypt.c
+++ b/drivers/md/dm-crypt.c
@@ -690,6 +690,8 @@ bad3:
bad2:
    crypto_free_tfm(tfm);
bad1:
+ /* Must zero key material before freeing */
+ memset(cc, 0, sizeof(*cc) + cc->key_size * sizeof(u8));
+ kfree(cc);
    return -EINVAL;
}
@@ -706,6 +708,9 @@ static void crypt_dtr(...)
    cc->iv_gen_ops->dtr(cc);
    crypto_free_tfm(cc->tfm);
    dm_put_device(ti, cc->dev);
+
+ /* Must zero key material before freeing */
+ memset(cc, 0, sizeof(*cc) + cc->key_size * sizeof(u8));
+ kfree(cc);
}

```

Listing 1: A security patch to fix an information disclosure vulnerability (CVE-2006-0095) in the Linux kernel.

considered—examples documented in the literature include manual control migration [40, 48], adaptive function cloning [58], and stack reconstruction [57].

We illustrate the state transfer problem with two update examples. Listing 1 presents a real-world security patch which fixes an information disclosure vulnerability (detailed in CVE-2006-0095 [5]) in the *md* (Multiple Device) driver of the Linux kernel. We sampled this patch from the dataset [3] originally used to evaluate Ksplice [11]. Similar to many other common security fixes, the patch considered introduces simple code changes that have no direct impact on the program state. The only tangible effect is the secure deallocation [24] of sensitive information on cryptographic keys. As a result, no state transformations are required at live update time. For this reason, Ksplice [11]—and other similar in-place live update tools—can deploy this update online with no state transfer necessary, allowing the new version to reuse the existing program state as is. Redirecting function invocations to the updated functions and resuming execution is sufficient to deploy the live update.

Listing 2 presents a sample patch providing a reduced test case for common code and data changes found in real-world updates. The patch introduces a number of type changes affecting a global **struct** variable (i.e., **var**)—with fields changed, removed, and reordered—and the necessary code changes to initialize the new data structure. Since the update significantly changes the in-memory representation of the global variable **var**, state transfer—using either automatically generated mapping functions or programmer-provided code—is necessary to transform the existing program state into a state compatible with the new version at live update time. Failure to do so would leave the new version in an invalid state after resuming execution. Section 5 shows how our state

```

--- a/example.c
+++ b/example.c
@@ -1,13 +1,12 @@
struct s {
    int count;
-   char str[3];
-   short id;
+   int id;
+   char str[2];
    union u u;
-   void *ptr;
    int addr;
-   short *inner_ptr;
+   int *inner_ptr;
} var;

void example_init(char *str) {
-   snprintf(var.str, 3, "%s", str);
+   snprintf(var.str, 2, "%s", str);
}

```

Listing 2: A sample patch introducing code and data changes that require state transfer at live update time.

transfer strategy can effectively automate this particular update, while traditional live update tools would largely delegate this major effort to the programmer.

State transfer has already been recognized as a hard problem in the literature. Qualitatively, many researchers have described it as “*tedious implementation of the transfer code*” [13], “*tedious engineering efforts*” [22], “*tedious work*” [23]. Others have discussed speculative [14, 16, 37, 38] and practical [63] ST scenarios which are particularly challenging (or unsolvable) even with programmer intervention. Quantitatively, a number of user-level live update tools for C programs (Ginseng [64], STUMP [62], and Kitsune [48]) have evaluated the ST manual effort in terms of lines of code (LOC). Table 1 presents a comparative analysis, with the number of updates analyzed, initial source changes to implement their live update mechanisms (LU LOC), and extra LOC to apply all the updates considered (ST LOC). In the last column, we report a normalized ST impact factor (Norm ST IF), measured as the expected ST LOC necessary after 100 updates normalized against the initial LU LOC.

As the table shows, the measured impacts are comparable (the lower impact in Kitsune stems from the greater initial annotation effort required by program-level updates) and demonstrate that ST increasingly (and heavily) dominates the manual effort in long-term deploy-

	#Upd	LU LOC	ST LOC	Norm ST IF
Ginseng	30	140	336	8.0x
STUMP	13	186	173	7.1x
Kitsune	40	523	554	2.6x

Table 1: State transfer impact (normalized after 100 updates) for existing user-level solutions for C programs.

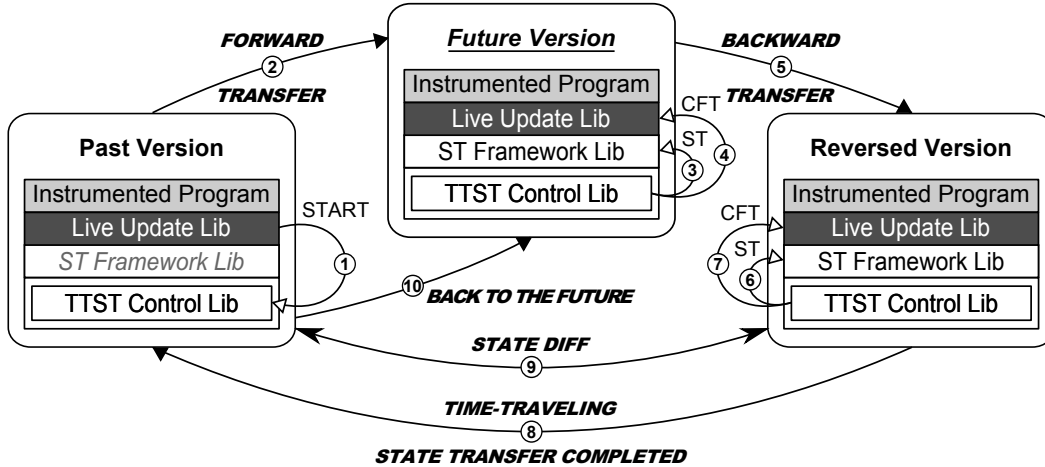


Figure 1: Time-traveling state transfer overview. The numbered arrows indicate the order of operations.

ment. Worse yet, any LOC-based metric underestimates the real ST effort, ignoring the atypical and error-prone programming model with nonstandard entry points, unconventional data access, and reduced testability and debuggability. Our investigation motivates our focus on automating and validating the state transfer process.

3 System Overview

We have designed our TTST live update technique with portability, extensibility, and interoperability in mind. This vision is reflected in our modular architecture, which enforces a strict separation of concerns and can support several possible live update tools and state transfer implementations. To use TTST, users need to statically instrument the target program in preparation for state transfer. In our current prototype, this is accomplished by a link-time transformation pass implemented using the LLVM compiler framework [56], which guarantees pain-free integration with existing GNU build systems using standard `configure` flags. We envision developers of the original program (i.e., users of our TTST technique) to gradually integrate support for our instrumentation into their development model, thus releasing live update-enabled software versions that can be easily managed by system administrators using simple tools. For this purpose, our TTST prototype includes `ttst-ctl`, a simple command-line tool that transparently interacts with the running program and allows system administrators to deploy live updates using our TTST technique with minimal effort. This can be simply done by using the following command-line syntax:

```
$ ttst-ctl `pidof program` ./new.bin
```

Runtime update functionalities, in turn, are implemented by three distinct libraries, transparently linked with the target program as part of our instrumentation

process. The *live update library* implements the update mechanisms specific to the particular live update tool considered. In detail, the library is responsible to provide the necessary update timing mechanisms [46] (e.g., start the live update when the program is *quiescent* [46] and all the external events are blocked) and CFT implementation. The *ST framework library*, in turn, implements the logic needed to automate state transfer and accommodate user-provided ST code. The *TTST control library*, finally, implements the resulting *time-traveling state transfer* process, with all the necessary mechanisms to coordinate the different process versions involved.

Our TTST technique operates across three process instances. The first is the original instance running the old software version (*past version*, from now on). This instance initiates, controls, and monitors the live update process, in particular running the only trusted library code in our architecture with respect to our fault model (§4). The second is a newly created instance running the new software version (*future version*, from now on). This instance is instructed to reinitialize its state from the past version. The third process instance is a clone of the past version created at live update time (*reversed version*, from now on). This instance is instructed to reinitialize its state from the future version. Figure 1 depicts the resulting architecture and live update process.

As shown in the figure, the update process is started by the live update library in the past version. This happens when the library detects that an update is available and all the necessary update timing restrictions (e.g., *quiescence* [46]) are met. The *start* event is delivered to the past version’s TTST control library, which sets out to initiate the *time-traveling state transfer* transaction. First, the library locates the new program version on the file system and creates the process instances for the future and reversed versions. Next, control is given to the future version’s TTST control library, requesting to

complete a *forward* state transfer run from the past version. In response, the library instructs the live update and ST framework libraries to perform ST and CFT, respectively. At the end of the process, control is given to the reversed version, where the TTST control library repeats the same steps to complete a *backward* state transfer run from the future version. Finally, the library notifies back the past version, where the TTST control library is waiting for TTST events. In response, the library performs *state differencing* between the past and reversed version to validate the TTST transaction and detect state corruption errors violating the semantics-preserving nature of the transformation. In our fault model, the past version is always immutable and adopted as an oracle when comparing the states. If the state is successfully validated (i.e., the past and reversed versions are identical), control moves *back to the future* version to resume execution. The other processes are automatically cleaned up.

When state corruption or run-time errors (e.g., crashes) are detected during the TTST transaction, the update is immediately aborted with the past version cleaning up the other instances and immediately resuming execution. The immutability of the past version's state allows the execution to resume exactly in the same state as it was right before the live update process started. This property ensures instant and transparent recovery in case of arbitrary TTST errors. Our recovery strategy enables fast and automated offline validation and, more importantly, a fault-tolerant live update process that can immediately and automatically rollback failed update attempts with no consequences for the running program.

4 Time-traveling State Transfer

The goal of TTST is to support a truly fault-tolerant live update process, which can automatically detect and recover from as many programming errors as possible, seamlessly support several live update tools and state transfer implementations, and rely on a minimal amount of trusted code at update time. To address these challenges, our TTST technique follows a number of key principles: a well-defined *fault model*, a large *state validation surface*, a *blackbox validation* strategy, and a generic *state transfer interface*.

Fault model. TTST assumes a general fault model with the ability to detect and recover from arbitrary *run-time* errors and *memory* errors introducing state corruption. In particular, run-time errors in the future and reversed versions are automatically detected by the TTST control library in the past version. The process abstraction allows the library to intercept abnormal termination errors in the other instances (e.g., crashes, panics) using simple tracing. Synchronization errors and infinite loops that prevent the TTST transaction from making progress,

in turn, are detected with a configurable update timeout (5s by default). Memory errors, finally, are detected by state differencing at the end of the TTST process.

Our focus on memory errors is motivated by three key observations. First, these represent an important class of nonsemantic state transfer errors, the only errors we can hope to detect in a fully automated fashion. Gupta's formal framework has already dismissed the possibility to automatically detect semantic state transfer errors in the general case [41]. Unlike memory errors, semantic errors are consistently introduced across forward and backward state transfer runs and thus cannot automatically be detected by our technique. As an example, consider an update that operates a simple semantic change: renumbering all the global error codes to use different value ranges. If the user does not explicitly provide additional ST code to perform the conversion, the default ST strategy will preserve the same (wrong) error codes across the future and the reversed version, with state differencing unable to detect any errors in the process.

Second, memory errors can lead to insidious latent bugs [32]—which can cause silent data corruption and manifest themselves potentially much later—or even introduce security vulnerabilities. These errors are particularly hard to detect and can easily escape the specification-based validation strategies adopted by all the existing live update testing tools [45–47].

Third, memory errors are painfully common in pathologically type-unsafe contexts like state transfer, where the program state is treated as an opaque object which must be potentially reconstructed from the ground up, all relying on the sole knowledge available to the particular state transfer implementation adopted.

Finally, note that, while other semantic ST errors cannot be detected in the general case, this does not preclude individual ST implementations from using additional knowledge to automatically detect some classes of errors in this category. For example, our state transfer framework can detect all the semantic errors that violate automatically derived *program state invariants* [33] (§5).

State validation surface. TTST seeks to validate the largest possible portion of the state, including state objects (e.g., global variables) that may only be accessed much later after the live update. To meet this goal, our state differencing strategy requires valid forward and backward transfer functions for each state object to validate. Clearly, the existence and the properties of such functions for every particular state object are subject to the nature of the update. For example, an update dropping a global variable in the new version has no defined backward transfer function for that variable. In other cases, forward and backward transfer functions exist but cannot be automatically generated. Consider the error code renumbering update exemplified earlier. Both

State	Diff	Fwd ST	Bwd ST	Detected
Unchanged	✓	STF	STF	Auto
Structural chg	✓	STF	STF	Auto
Semantic chg	✓	User	User ¹	Auto ¹
Dropped	✓	-	-	Auto
Added	✗	Auto/User	-	STF

¹Optional

Table 2: State validation and error detection surface.

the forward and backward transfer functions for all the global variables affected would have to be manually provided by the user. Since we wish to support fully automated validation by default (mandating extra manual effort is likely to discourage adoption), we allow TTST to gracefully reduce the state validation surface when backward transfer functions are missing—without hampering the effectiveness of our strategy on other fully transferable state objects. Enforcing this behavior in our design is straightforward: the reversed version is originally cloned from the past version and all the state objects that do not take part in the backward state transfer run will trivially match their original counterparts in the state differencing process (unless state corruption occurs).

Table 2 analyzes TTST’s state validation and error detection surface for the possible state changes introduced by a given update. The first column refers to the nature of the transformation of a particular state object. The second column refers to the ability to validate the state object using state differencing. The third and fourth column characterize the implementation of the resulting forward and backward transfer functions. Finally, the fifth column analyzes the effectiveness in detecting state corruption. For unchanged state objects, state differencing can automatically detect state corruption and transfer functions are automatically provided by the state transfer framework (STF). Note that unchanged state objects do not necessarily have the same representation in the different versions. The memory layout of an updated version does not generally reflect the memory layout of the old version and the presence of pointers can introduce representation differences for some unchanged state objects between the past and future version. State objects with structural changes exhibit similar behavior, with a fully automated transfer and validation strategy. With structural changes, we refer to state changes that affect only the type representation and can be entirely arbitrated from the STF with no user intervention (§5). This is in contrast with semantic changes, which require user-provided transfer code and can only be partially automated by the STF (§5). Semantic state changes highlight the tradeoff between state validation coverage and the manual effort required by the user. In a traditional

live update scenario, the user would normally only provide a forward transfer function. This behavior is seamlessly supported by TTST, but the transferred state object will not be considered for validation. If the user provides code for the reverse transformation, however, the transfer can be normally validated with no restriction. In addition, the backward transfer function provided can be used to perform a cold rollback from the future version to the past version (i.e., live updating the new version into the old version at a later time, for example when the administrator experiences an unacceptable performance slowdown in the updated version). Dropped state objects, in turn, do not require any explicit transfer functions and are automatically validated by state differencing as discussed earlier. State objects that are added in the update (e.g., a new global variable), finally, cannot be automatically validated by state differencing and their validation and transfer is delegated to the STF (§5) or to the user.

Blackbox validation. TTST follows a blackbox validation model, which completely ignores ST internals. This is important for two reasons. First, this provides the ability to support many possible updates and ST implementations. This also allows one to evaluate and compare different STFs. Second, this is crucial to decouple the validation logic from the ST implementation, minimizing the amount of trusted code required by our strategy. In particular, our design goals dictate the minimization of the *reliable computing base (RCB)*, defined as the core software components that are necessary to ensure correct implementation behavior [26]. Our fault model requires four primary components in the RCB: the update timing mechanisms, the TTST arbitration logic, the run-time error detection mechanisms, and the state differencing logic. All the other software components which run in the future and reversed versions (e.g., ST code and CFT code) are fully untrusted thanks to our design.

The implementation of the update timing mechanisms is entirely delegated to the live update library and its size subject to the particular live update tool considered. We trust that every reasonable update timing implementation will have a small RCB impact. For the other TTST components, we seek to reduce the code size (and complexity) to the minimum. Luckily, our TTST arbitration logic and run-time error detection mechanisms (described earlier) are straightforward and only marginally contribute to the RCB. In addition, TTST’s semantics-preserving ST transaction and structural equivalence between the final (reversed) state and the original (past) state ensure that the memory images of the two versions are always identical in error-free ST runs. This drastically simplifies our state differencing strategy, which can be implemented using trivial word-by-word memory comparison, with no other knowledge on the ST code and marginal RCB impact. Our comparison strategy examines all the

```

function STATE_DIFF(pid1, pid2)
  a ← addr_start
  while a < shadow_start do
    m1 ← IS_MAPPED_WRITABLE(a, pid1)
    m2 ← IS_MAPPED_WRITABLE(a, pid2)
    if m1 or m2 then
      if m1 ≠ m2 then
        return true
      if MEMPAGECMP(a, pid1, pid2) ≠ 0 then
        return true
    a ← a + page_size
  return false

```

Figure 2: State differencing pseudocode.

writable regions of the address space excluding only private shadow stack/heap regions (mapped at the end of the address space) in use by the TTST control library. Figure 2 shows the pseudocode for this simple strategy.

State transfer interface. TTST’s state transfer interface seeks to minimize the requirements and the effort to implement the STF. In terms of requirements, TTST demands only a *layout-aware* and *user-aware* STF semantic. By layout-aware, we refer to the ability of the STF to preserve the original state layout when requested (i.e., in the reversed version), as well as to automatically identify the state changes described in Table 2. By user-aware, we refer to the ability to allow the user to selectively specify new forward and backward transfer functions and candidate state objects for validation. To reduce the effort, TTST offers a convenient STF programming model, with an error handling-friendly environment—our fault-tolerant design encourages indiscriminated use of assertions—and a generic interprocess communication (IPC) interface. In particular, TTST implements an IPC *control* interface to coordinate the TTST transaction and an IPC *data* interface to grant read-only access to the state of a given process version to the others. These interfaces are currently implemented by UNIX domain sockets and POSIX shared memory (respectively), but other IPC mechanisms can be easily supported. The current implementation combines fast data transfer with a secure design that prevents impersonation attacks (access is granted only to the predetermined process instances).

5 State Transfer Framework

Our state transfer framework seeks to automate all the possible ST steps, leaving only the undecidable cases (e.g., semantic state changes) to the user. The implementation described here optimizes and extends our prior work [33–36] to the TTST model. We propose a STF design that resembles a *moving*, *mutating*, and *interpro-*

cess garbage collection model. By moving, we refer to the ability to relocate (and possibly reallocate) static and dynamic state objects in the next version. This is to allow arbitrary changes in the memory layout between versions. By mutating, we refer to the ability to perform on-the-fly type transformations when transferring every given state object from the previous to the next version. Interprocess, finally, refers to our process-level ST strategy. Our goals raise 3 major challenges for a low-level language like C. First, our moving requirement requires precise object and pointer analysis at runtime. Second, on-the-fly type transformations require the ability to dynamically identify, inspect, and match generic data types. Finally, our interprocess strategy requires a mechanism to identify and map state objects across process versions.

Overview. To meet our goals, our STF uses a combination of static and dynamic ST instrumentation. Our static instrumentation, implemented by a LLVM link-time pass [56], transforms each program version to generate *metadata* information that surgically describes the entirety of the program state. In particular, static metadata, which provides *relocation* and *type* information for all the static state objects (e.g., global variables, strings, functions with address taken), is embedded directly into the final binary. Dynamic metadata, which provides the same information for all the dynamic state objects (e.g., heap-allocated objects), is, in turn, dynamically generated/destroyed at runtime by our allocation/deallocation site instrumentation—we currently support `malloc/mmap-like` allocators automatically and standard region-based allocators [15] using user-annotated allocator functions. Further, our pass can dynamically generate/destroy local variable metadata for a predetermined number of functions (e.g., `main`), as dictated by the particular update model considered. Finally, to automatically identify and map objects across process versions, our instrumentation relies on *version-agnostic* state IDs derived from unambiguous *naming* and *contextual* information. In detail, every static object is assigned a static ID derived by its source name (e.g., function name) and scope (e.g., static variable module). Every dynamic object, in turn, is assigned a static ID derived by allocation site information (e.g., caller function name and target pointer name) and an incremental dynamic ID to unambiguously identify allocations at runtime.

Our ID-based naming scheme fulfills TTST’s layout-awareness goal: static IDs are used to identify state changes and to automatically reallocate dynamic objects in the future version; dynamic IDs are used to map dynamic objects in the future version with their existing counterparts in the reversed version. The mapping policy to use is specified as part of generic *ST policies*, also implementing other TTST-aware extensions: (i) *randomization* (enabled in the future version): perform

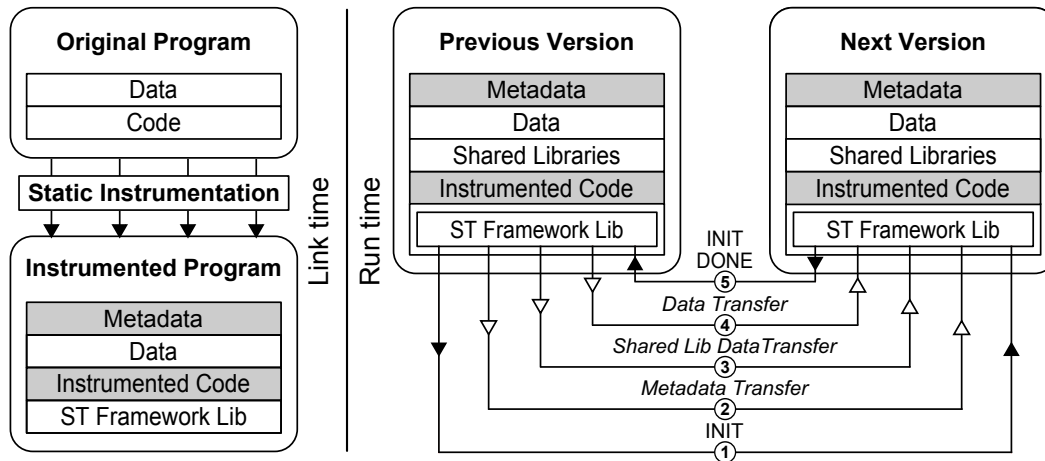


Figure 3: State transfer framework overview.

fine-grained address space randomization [34] for all the static/dynamically reallocated objects, used to amplify the difference introduced by memory errors in the overall TTST transaction; (ii) *validation* (enabled in the reversed version): zero out the local copy of all the mapped state objects scheduled for automated transfer to detect missing write errors at validation time.

Our dynamic instrumentation, included in a preloaded shared library (ST framework library), complements the static pass to address the necessary run-time tasks: type and pointer analysis, metadata management for shared libraries, error detection. In addition, the ST framework library implements all the steps of the ST process, as depicted in Figure 3. The process begins with an initialization request from the TTST control library, which specifies the ST policies and provides access to the TTST’s IPC interface. The next *metadata transfer* step transfers all the metadata information from the previous version to a metadata cache in the next version (local address space). At the end, the local state objects (and their metadata) are mapped into the external objects described by the metadata cache and scheduled for transfer according to their state IDs and the given ST policies. The next two *data transfer* steps complete the ST process, transferring all the data to reinitialize shared library and program state to the next version. State objects scheduled for transfer are processed one at a time, using metadata information to locate the objects and their internal representations in the two process versions and apply pointer and type transformations on the fly. The last step performs cleanup tasks and returns control to the caller.

State transfer strategy. Our STF follows a well-defined automated ST strategy for all the mapped state objects scheduled for transfer, exemplified in Figure 4. As shown in the figure—which reprises the update example given earlier (§ 2)—our type analysis automatically and recursively matches individual type elements be-

tween object versions by *name* and *representation*, identifying added/dropped/changed/identical elements on the fly. This strategy automates ST for common structural changes, including: primitive type changes, array expansion/truncation, and addition/deletion/reordering of **struct** members. Our pointer analysis, in turn, implements a generic pointer transfer strategy, automatically identifying (base and interior) pointer targets in the previous version and reinitializing the pointer values correctly in the next version, in spite of type and memory layout changes. To perform efficient pointer lookups, our analysis organizes all the state objects with address taken in a splay tree, an idea previously explored by bounds checkers [9, 27, 70]. We also support all the special pointer idioms allowed by C (e.g., guard pointers) automatically, with the exception of cases of “*pointer ambiguity*” [36].

To deal with ambiguous pointer scenarios (e.g., **unions** with inner pointers and pointers stored as integers) as well as more complex state changes (e.g., semantic changes), our STF supports user extensions in the form of preprocessor annotations and callbacks. Figure 4 shows an example of two ST annotations: **IXFER** (force memory copying with no pointer transfer) and **PXFER** (force pointer transfer instead of memory copying). Callbacks, in turn, are evaluated whenever the STF maps or traverses a given object or type element, allowing the user to override the default mapping behavior (e.g., for renamed variables) or express sophisticated state transformations at the object or element level. Callbacks can be also used to: (i) override the default validation policies, (ii) initialize new state objects; (iii) instruct the STF to checksum new state objects after initialization to detect memory errors at the end of the ST process.

Shared libraries. Uninstrumented shared libraries (SLs) pose a major challenge to our pointer transfer strategy. In particular, failure to reinitialize SL-related pointers correctly in the future version would introduce er-

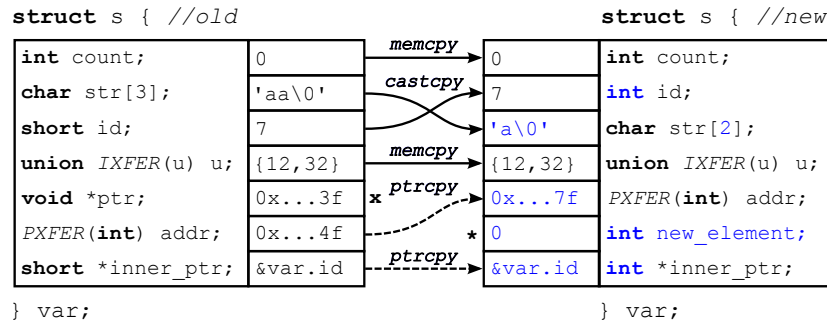


Figure 4: Automated state transfer example for the data structure presented in Listing 2.

rors after live update. To address this challenge, our STF distinguishes 3 scenarios: (i) program/SL pointers into static SL state; (ii) program/SL pointers into dynamic SL state; (iii) SL pointers into static or dynamic program state. To deal with the first scenario, our STF instructs the dynamic linker to remap all the SLs in the future version at the same addresses as in the past version, allowing SL data transfer (pointer transfer in particular) to be implemented via simple memory copying. SL relocation is currently accomplished by prelinking the SLs on demand when starting the future version, a strategy similar to “*retouching*” for mobile applications [19]. To address the second scenario, our dynamic instrumentation intercepts all the memory management calls performed by SLs and generates dedicated metadata to reallocate the resulting objects at the same address in the future version. This is done by restoring the original heap layout (and content) as part of the SL data transfer phase. To perform heap randomization and type transformations correctly for all the program allocations in the future version, in turn, we allow the STF to deallocate (and reallocate later) all the non-SL heap allocations right after SL data transfer. To deal with the last scenario, we need to accurately identify all the SL pointers into the program state and update their values correctly to reflect the memory layout of the future version. Luckily, these cases are rare and we can envision library developers exporting a public API that clearly marks long-lived pointers into the program state once our live update technique is deployed. A similar API is desirable to mark all the process-specific state (e.g., *libc*’s cached pids) that should be restored after ST—note that shareable resources like file descriptors are, in contrast, automatically transferred by the `fork/exec` paradigm. To automate the identification of these cases in our current prototype, we used conservative pointer analysis techniques [17, 18] under stress testing to locate long-lived SL pointers into the program state and state differencing at `fork` points to locate process-specific state objects.

Error detection. To detect certain classes of semantic errors that escape TTST’s detection strategy, our

STF enforces *program state invariants* [33] derived from all the metadata available at runtime. Unlike existing *likely* invariant-based error detection techniques [6, 28, 31, 42, 68], our invariants are conservatively computed from static analysis and allow for no false positives. The majority of our invariants are enforced by our dynamic pointer analysis to detect semantic errors during pointer transfer. For example, our STF reports invariant violation (and aborts ST by default) whenever a pointer target no longer exists or has its address taken (according to our static analysis) in the new version. Another example is a transferred pointer that points to an illegal target type according to our static pointer cast analysis.

6 Evaluation

We have implemented TTST on Linux (x86), with support for generic user-space C programs using the ELF binary format. All the platform-specific components, however, are well isolated in the TTST control library and easily portable to other operating systems, architectures, and binary formats other than ELF. We have integrated address space randomization techniques developed in prior work [34] into our ST instrumentation and configured them to randomize the location of all the static and dynamically reallocated objects in the future version. To evaluate TTST, we have also developed a live update library mimicking the behavior of state-of-the-art live update tools [48], which required implementing preannotated per-thread update points to control update timing, manual control migration to perform CFT, and a UNIX domain sockets-based interface to receive live update commands from our `ttst-ctl` tool.

We evaluated the resulting solution on a workstation running Linux v3.5.0 (x86) and equipped with a 4-core 3.0Ghz AMD Phenom II X4 B95 processor and 8GB of RAM. For our evaluation, we first selected Apache httpd (v2.2.23) and nginx (v0.8.54), the two most popular open-source web servers. For comparison purposes, we also considered vsftpd (v1.1.0) and the OpenSSH dae-

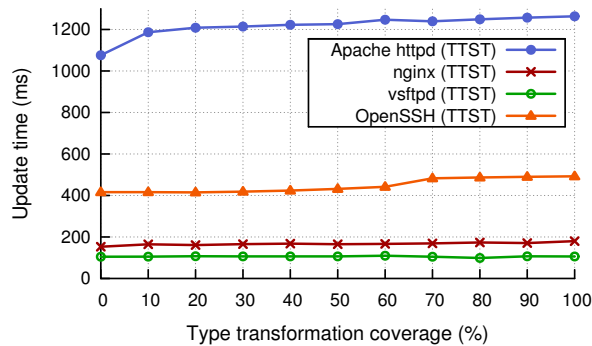


Figure 5: Update time vs. type transformation coverage.

mon (v3.5p1), a popular open-source ftp and ssh server, respectively. The former [23,45,48,49,57,63,64] and the latter [23,45,64] are by far the most used server programs (and versions) in prior work in the field. We annotated all the programs considered to match the implemented live update library as described in prior work [45,48]. For Apache httpd and nginx, we redirected all the calls to custom allocation routines to the standard allocator interface (i.e., `malloc/free` calls), given that our current instrumentation does not yet support custom allocation schemes based on nested regions [15] (Apache httpd) and slab-like allocations [20] (nginx). To evaluate our programs, we performed tests using the Apache benchmark (AB) [1] (Apache httpd and nginx), dkftpbench [2] (vsftpd), and the provided regression test suite (OpenSSH). We configured our programs and benchmarks using the default settings. We repeated all our experiments 21 times and reported the median—with negligible standard deviation measured across multiple test runs.

Our evaluation answers five key questions: (i) *Performance*: Does TTST yield low run-time overhead and reasonable update times? (ii) *Memory usage*: How much memory do our instrumentation techniques use? (iii) *RCB size*: How much code is (and is not) in the RCB? (iv) *Fault tolerance*: Can TTST withstand arbitrary failures in our fault model? (v) *Engineering effort*: How much engineering effort is required to adopt TTST?

Performance. To evaluate the run-time overhead imposed by our update mechanisms, we first ran our benchmarks to compare our base programs with their instrumented and annotated versions. Our experiments showed no appreciable performance degradation. This is expected, since update points only require checking a flag at the top of long-running loops and metadata is efficiently managed by our ST instrumentation. In detail, our static metadata—used only at update time—is confined in a separate ELF section so as not to disrupt locality. Dynamic metadata management, in turn, relies on in-band descriptors to minimize the overhead

Type	httpd	nginx	vsftpd	OpenSSH
Static	2.187	2.358	3.352	2.480
Run-time	3.100	3.786	4.362	2.662
Forward ST	3.134	5.563	6.196	4.126
TTST	3.167	7.340	8.031	5.590

Table 3: TTST-induced memory usage (measured statically or at runtime) normalized against the baseline.

on allocator operations. To evaluate the latter, we instrumented all the C programs in the SPEC CPU2006 benchmark suite. The results evidenced a 4% average run-time overhead across all the benchmarks. We also measured the cost of our instrumentation on 10,000 `malloc/free` and `mmap/munmap` repeated `glibc` allocator operations—which provide worst-case results, given that common allocation patterns generally yield poorer locality. Experiments with multiple allocation sizes (0-16MB) reported a maximum overhead of 41% for `malloc`, 9% for `free`, 77% for `mmap`, and 42% for `munmap`. While these microbenchmark results are useful to evaluate the impact of our instrumentation on allocator operations, we expect any overhead to be hardly visible in real-world server programs, which already strive to avoid expensive allocations on the critical path [15].

When compared to prior user-level solutions, our performance overhead is much lower than more intrusive instrumentation strategies—with worst-case macrobenchmark overhead of 6% [64], 6.71% [62], and 96.4% [57]—and generally higher than simple binary rewriting strategies [10, 23]—with worst-case function invocation overhead estimated around 8% [58]. Unlike prior solutions, however, our overhead is strictly isolated in allocator operations and never increases with the number of live updates deployed over time. Recent program-level solutions that use minimal instrumentation [48]—no allocator instrumentation, in particular—in turn, report even lower overheads than ours, but at the daunting cost of annotating all the pointers into heap objects.

We also analyzed the impact of process-level TTST on the update time—the time from the moment the update is signaled to the moment the future version resumes execution. Figure 5 depicts the update time—when updating the master process of each program—as a function of the number of type transformations operated by our ST framework. For this experiment, we implemented a source-to-source transformation able to automatically change 0-1,327 type definitions (adding/reordering `struct` fields and expanding arrays/primitive types) for Apache httpd, 0-818 type definitions for nginx, 0-142 type definitions for vsftpd, and 0-455 type definitions for OpenSSH between versions. This forced our ST framework to operate an average of 1,143,981, 111,707,

Component	RCB	Other
ST instrumentation	1,119	8,211
Live update library	235	147
TTST control library	412	2,797
ST framework	0	13,311
ttst-ctl tool	0	381
Total	1,766	26,613

Table 4: Source lines of code (LOC) and contribution to the RCB size for every component in our architecture.

1,372, and 206,259 type transformations (respectively) at 100% coverage. As the figure shows, the number of type transformations has a steady but low impact on the update time, confirming that the latter is heavily dominated by memory copying and pointer analysis—albeit optimized with splay trees. The data points at 100% coverage, however, are a useful indication of the upper bound for the update time, resulting in 1263 ms, 180 ms, 112 ms, and 465 ms (respectively) with our TTST update strategy. Apache httpd reported the longest update times in all the configurations, given the greater amount of state transferred at update time. Further, TTST update times are, on average, 1.76x higher than regular ST updates (not shown in figure for clarity), acknowledging the impact of backward ST and state differencing on the update time. While our update times are generally higher than prior solutions, the impact is bearable for most programs and the benefit is stateful *fault-tolerant* version updates.

Memory usage. Our state transfer instrumentation leads to larger binary sizes and run-time memory footprints. This stems from our metadata generation strategy and the libraries required to support live update. Table 3 evaluates the impact on our test programs. The static memory overhead (235.2% worst-case overhead for vsftpd) measures the impact of our ST instrumentation on the binary size. The run-time overhead (336.2% worst-case overhead for vsftpd), in turn, measures the impact of instrumentation and support libraries on the virtual memory size observed at runtime, right after server initialization. These measurements have been obtained starting from a baseline virtual memory size of 234 MB for Apache httpd and less than 6 MB for all the other programs. The third and the fourth rows, finally, show the maximum virtual memory overhead we observed at live update time for both regular (forward ST only) and TTST updates, also accounting for all the transient process instances created (703.1% worst-case overhead for vsftpd and TTST updates). While clearly program-dependent and generally higher than prior live update solutions, our measured memory overheads are modest and, we believe, realistic for most systems, also given the increasingly low cost of RAM in these days.

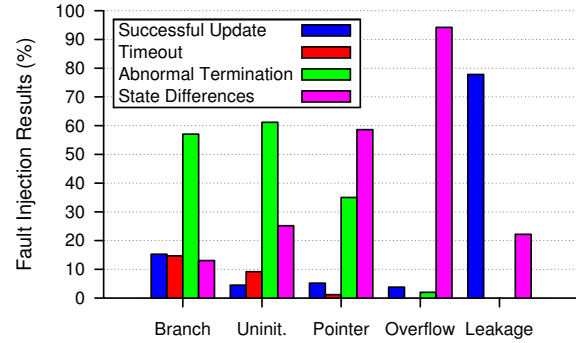


Figure 6: TTST behavior in our automated fault injection experiments for varying fault types.

RCB size. Our TTST update technique is carefully designed to minimize the RCB size. Table 4 lists the LOC required to implement every component in our architecture and the contributions to the RCB. Our ST instrumentation requires 1,119 RCB LOC to perform dynamic metadata management at runtime. Our live update library requires 235 RCB LOC to implement the update timing mechanisms and interactions with client tools. Our TTST control library requires 412 RCB LOC to arbitrate the TTST process, implement run-time error detection, and perform state differencing—all from the past version. Our ST framework and ttst-ctl tool, in contrast, make no contribution to the RCB. Overall, our design is effective in producing a small RCB, with only 1,766 LOC compared to the other 26,613 non-RCB LOC. Encouragingly, our RCB is even substantially smaller than that of other systems that have already been shown to be amenable to formal verification [54]. This is in stark contrast with all the prior solutions, which make no effort to remove *any* code from the RCB.

Fault tolerance. We evaluated the fault tolerance of TTST using software-implemented fault injection (SWIFI) experiments. To this end, we implemented another LLVM pass which transforms the original program to inject specific classes of software faults into predetermined code regions. Our pass accepts a list of target program functions/modules, the fault types to inject, and a fault probability ϕ —which specifies how many fault locations should be randomly selected for injection out of all the possible candidates found in the code. We configured our pass to randomly inject faults in the ST code, selecting $\phi = 1\%$ —although we observed similar results for other ϕ values—and fault types that matched common programming errors in our fault model. In detail, similar to prior SWIFI strategies that evaluated the effectiveness of fault-tolerance mechanisms against state corruption [65], we considered generic *branch* errors (branch/loop condition flip or stuck-at errors) as well as

	Updates		Changes			Engineering effort		
	#	LOC	Fun	Var	Ty	ST Ann LOC	Fwd ST LOC	Bwd ST LOC
Apache httpd	5	10,844	829	28	48	79	302	151
nginx	25	9,681	711	51	54	24	335	0
vsftpd	5	5,830	305	121	35	0	21	21
OpenSSH	5	14,370	894	84	33	0	135	127
Total	40	40,725	2,739	284	170	103	793	299

Table 5: Engineering effort for all the updates analyzed in our evaluation.

common memory errors, such as *uninitialized reads* (emulated by missing initializers), *pointer corruption* (emulated by corrupting pointers with random or off-by-1 values), *buffer overflows* (emulated by extending the size passed to data copy functions, e.g., `memcpy`, by 1-100%), and *memory leakage* (emulated by missing deallocation calls). We repeated our experiments 500 times for each of the 5 fault types considered, with each run starting a live update between randomized program versions and reporting the outcome of our TTST strategy. We report results only for vsftpd—although we observed similar results for the other programs—which allowed us to collect the highest number of fault injection samples per time unit and thus obtain the most statistically sound results.

Figure 6 presents our results breaking down the data by fault type and distribution of the observed outcomes—that is, update succeeded or automatically rolled back after *timeout*, *abnormal termination* (e.g., crash), or past-reversed *state differences* detected. As expected, the distribution varies across the different fault types considered. For instance, branch and initialization errors produced the highest number of updates aborted after a timeout (14.6% and 9.2%), given the higher probability of infinite loops. The first three classes of errors considered, in turn, resulted in a high number of crashes (51.1%, on average), mostly due to invalid pointer dereferences and invariants violations detected by our ST framework. In many cases, however, the state corruption introduced did not prevent the ST process from running to completion, but was nonetheless detected by our state differencing technique. We were particularly impressed by the effectiveness of our validation strategy in a number of scenarios. For instance, state differencing was able to automatically recover from as many as 471 otherwise-unrecoverable buffer overflow errors. Similar is the case of memory leakages—actually activated in 22.2% of the runs—with any extra memory region mapped by our metadata cache and never deallocated immediately detected at state diffing time. We also verified that the future (or past) version resumed execution correctly after every successful (or aborted) update attempt. When sampling the 533 successful cases, we noted the introduction

of irrelevant faults (e.g., missing initializer for an unused variable) or no faults actually activated at runtime. Overall, our TTST technique was remarkably effective in detecting and recovering from a significant number of observed failures (1,967 overall), with no consequences for the running program. This is in stark contrast with all the prior solutions, which make *no* effort in this regard.

Engineering effort. To evaluate the engineering effort required to deploy TTST, we analyzed a number of official incremental releases following our original program versions and prepared the resulting patches for live update. In particular, we considered 5 updates for Apache httpd (v2.2.23-v2.3.8), vsftpd (v1.1.0-v2.0.2), and OpenSSH (v3.5-v3.8), and 25 updates for nginx (v0.8.54-v1.0.15), given that nginx’s tight release cycle generally produces incremental patches that are much smaller than those of the other programs considered. Table 5 presents our findings. The first two grouped columns provide an overview of our analysis, with the number of updates considered for each program and the number of lines of code (LOC) added, deleted, or modified in total by the updates. As shown in the table, we manually processed more than 40,000 LOC across the 40 updates considered. The second group shows the number of functions, variables, and types changed (i.e., added, deleted, or modified) by the updates, with a total of 2,739, 284, and 170 changes (respectively). The third group, finally, shows the engineering effort in terms of LOC required to prepare our test programs and our patches for live update. The first column shows the one-time annotation effort required to integrate our test programs with our ST framework. Apache httpd and nginx required 79 and 2 LOC to annotate 12 and 2 **unions** with inner pointers, respectively. In addition, nginx required 22 LOC to annotate a number of global pointers using special data encoding—storing metadata information in the 2 least significant bits. The latter is necessary to ensure precise pointer analysis at ST time. The second and the third column, in turn, show the number of lines of state transfer code we had to manually write to complete forward ST and backward ST (respectively) across all the updates considered. Such ST extensions were necessary

to implement complex state changes that could not be automatically handled by our ST framework.

A total of 793 forward ST LOC were strictly necessary to prepare our patches for live update. An extra 299 LOC, in turn, were required to implement backward ST. While optional, the latter is important to guarantee full validation surface for our TTST technique. The much lower LOC required for backward ST (37.7%) is easily explained by the additive nature of typical state changes, which frequently entail only adding new data structures (or fields) and thus rarely require extra LOC in our backward ST transformation. The case of *nginx* is particularly emblematic. Its disciplined update strategy, which limits the number of nonadditive state changes to the minimum, translated to no manual ST LOC required to implement backward ST. We believe this is particularly encouraging and can motivate developers to deploy our TTST techniques with full validation surface in practice.

7 Related Work

Live update systems. We focus on *local* live update solutions for generic and widely deployed C programs, referring the reader to [7, 8, 29, 55, 74] for distributed live update systems. LUCOS [22], DynaMOS [58], and Ksplice [11] have applied live updates to the Linux kernel, loading new code and data directly into the running version. Code changes are handled using binary rewriting (i.e., trampolines). Data changes are handled using shadow [11, 58] or parallel [22] data structures. OPUS [10], POLUS [23], Ginseng [64], STUMP [62], and Upstare [57] are similar live update solutions for user-space C programs. Code changes are handled using binary rewriting [10, 23], compiler-based instrumentation [62, 64], or stack reconstruction [57]. Data changes are handled using parallel data structures [23], type wrapping [62, 64], or object replacement [57]. Most solutions delegate ST entirely to the programmer [10, 11, 22, 23, 58], others generate only basic type transformers [57, 62, 64]. Unlike TTST, none of these solutions attempt to fully automate ST—pointer transfer, in particular—and state validation. Further, their in-place update model hampers isolation and recovery from ST errors, while also introducing address space fragmentation over time. To address these issues, alternative update models have been proposed. Prior work on process-level live updates [40, 49], however, delegates the ST burden entirely to the programmer. In another direction, Kitsune [48] encapsulates every program in a hot swappable shared library. Their state transfer framework, however, does not attempt to automate pointer transfer without user effort and no support is given to validate the state or perform safe rollback in case of ST errors. Finally, our prior work [34, 35] demonstrated the benefits of process-

level live updates in component-based OS architectures, with support to recover from run-time ST errors but no ability to detect a corrupted state in the updated version.

Live update safety. Prior work on live update safety is mainly concerned with safe update timing mechanisms, neglecting important system properties like fault tolerance and RCB minimization. Some solutions rely on *quiescence* [10–13] (i.e., no updates to active code), others enforce *representation consistency* [62, 64, 71] (i.e., no updated code accessing old data). Other researchers have proposed using transactions in local [63] or distributed [55, 74] contexts to enforce stronger timing constraints. Recent work [44], in contrast, suggests that many researchers may have been overly concerned with update timing and that a few predetermined update points [34, 35, 48, 49, 62, 64] are typically sufficient to determine safe and timely update states. Unlike TTST, none of the existing solutions have explicitly addressed ST-specific update safety properties. Static analysis proposed in OPUS [10]—to detect unsafe data updates—and Ginseng [64]—to detect unsafe pointers into updated objects—is somewhat related, but it is only useful to *disallow* particular classes of (unsupported) live updates.

Update testing. Prior work on live update testing [45–47] is mainly concerned with validating the correctness of an update in all the possible update timings. Correct execution is established from manually written specifications [47] or manually selected program output [45, 46]. Unlike TTST, these techniques require nontrivial manual effort, are only suitable for offline testing, and fail to validate the entirety of the program state. In detail, their state validation surface is subject to the coverage of the test programs or specifications used. Their testing strategy, however, is useful to compare different update timing mechanisms, as also demonstrated in [45]. Other related work includes online patch validation, which seeks to efficiently compare the behavior of two (original and patched) versions at runtime. This is accomplished by running two separate (synchronized) versions in parallel [21, 51, 59] or a single hybrid version using a split-and-merge strategy [73]. These efforts are complementary to our work, given that their goal is to test for errors in the patch itself rather than validating the state transfer code required to prepare the patch for live update. Complementary to our work are also efforts on upgrade testing in large-scale installations, which aim at creating sandboxed deployment-like environments for testing purposes [75] or efficiently testing upgrades in diverse environments using staged deployment [25]. Finally, fault injection has been previously used in the context of update testing [29, 60, 66], but only to emulate upgrade-time operator errors. Our evaluation, in contrast, presents the first fault injection campaign that emulates realistic programming errors in the ST code.

8 Conclusion

While long recognized as a hard problem, state transfer has received limited attention in the live update literature. Most efforts focus on automating and validating update timing, rather than simplifying and shielding the state transfer process from programming errors. We believe this is a key factor that has discouraged the system administration community from adopting live update tools, which are often deemed impractical and untrustworthy.

This paper presented *time-traveling state transfer*, the first fault-tolerant live update technique which allows generic live update tools for C programs to automate and validate the state transfer process. Our technique combines the conventional forward state transfer transformation with a backward (and logically redundant) transformation, resulting in a semantics-preserving manipulation of the original program state. Observed deviations in the reversed state are used to automatically identify state corruption caused by common classes of programming errors (i.e., memory errors) in the state transfer (library or user) code. Our process-level update strategy, in turn, guarantees detection of other run-time errors (e.g., crashes), simplifies state management, and prevents state transfer errors to propagate back to the original version. The latter property allows our framework to safely recover from errors and automatically resume execution in the original version. Further, our modular and blackbox validation design yields a minimal-RCB live update system, offering a high fault-tolerance surface in both online and offline validation runs. Finally, we complemented our techniques with a generic state transfer framework, which automates state transformations with minimal programming effort and can detect additional semantic errors using statically computed invariants. We see our work as the first important step toward truly practical and trustworthy live update tools for system administrators.

9 Acknowledgments

We would like to thank our shepherd, Mike Ciavarella, and the anonymous reviewers for their comments. This work has been supported by European Research Council under grant ERC Advanced Grant 2008 - R3S3.

References

- [1] Apache benchmark (AB). <http://httpd.apache.org/docs/2.0/programs/ab.html>.
- [2] dkftpbench. <http://www.kegel.com/dkftpbench>.
- [3] Ksplice performance on security patches. <http://www.ksplice.com/cve-evaluation>.
- [4] Ksplice Uptrack. <http://www.ksplice.com>.
- [5] Vulnerability summary for CVE-2006-0095. <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2006-0095>.
- [6] ADVE, S. V., ADVE, V. S., AND ZHOU, Y. Using likely program invariants to detect hardware errors. In *Proc. of the IEEE Int'l Conf. on Dependable Systems and Networks* (2008).
- [7] AJMANI, S., LISKOV, B., AND SHRIRA, L. Scheduling and simulation: How to upgrade distributed systems. In *Proc. of the Ninth Workshop on Hot Topics in Operating Systems* (2003), vol. 9, pp. 43–48.
- [8] AJMANI, S., LISKOV, B., SHRIRA, L., AND THOMAS, D. Modular software upgrades for distributed systems. In *Proc. of the 20th European Conf. on Object-Oriented Programming* (2006), pp. 452–476.
- [9] AKRITIDIS, P., COSTA, M., CASTRO, M., AND HAND, S. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proc. of the 18th USENIX Security Symp.* (2009), pp. 51–66.
- [10] ALTEKAR, G., BAGRAK, I., BURSTEIN, P., AND SCHULTZ, A. OPUS: Online patches and updates for security. In *Proc. of the 14th USENIX Security Symp.* (2005), vol. 14, pp. 19–19.
- [11] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: Automatic rebootless kernel updates. In *Proc. of the Fourth ACM European Conf. on Computer Systems* (2009), pp. 187–198.
- [12] BAUMANN, A., APPAVOO, J., WISNIEWSKI, R. W., SILVA, D. D., KRIEGER, O., AND HEISER, G. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *Proc. of the USENIX Annual Tech. Conf.* (2007), pp. 1–14.
- [13] BAUMANN, A., HEISER, G., APPAVOO, J., DA SILVA, D., KRIEGER, O., WISNIEWSKI, R. W., AND KERR, J. Providing dynamic update in an operating system. In *Proc. of the USENIX Annual Tech. Conf.* (2005), p. 32.
- [14] BAZZI, R. A., MAKRIKIS, K., NAYERI, P., AND SHEN, J. Dynamic software updates: The state mapping problem. In *Proc. of the Second Int'l Workshop on Hot Topics in Software Upgrades* (2009), p. 2.
- [15] BERGER, E. D., ZORN, B. G., AND MCKINLEY, K. S. Reconsidering custom memory allocation. In *Proc. of the 17th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications* (2002), pp. 1–12.
- [16] BLOOM, T., AND DAY, M. Reconfiguration and module replacement in Argus: Theory and practice. *Software Engineering J.* 8, 2 (1993), 102–108.
- [17] BOEHM, H.-J. Space efficient conservative garbage collection. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation* (1993), pp. 197–206.
- [18] BOEHM, H.-J. Bounding space usage of conservative garbage collectors. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (2002), pp. 93–100.

- [19] BOJINOV, H., BONEH, D., CANNINGS, R., AND MALCHEV, I. Address space randomization for mobile devices. In *Proc. of the Fourth ACM Conf. on Wireless network security* (2011), pp. 127–138.
- [20] BONWICK, J. The slab allocator: An object-caching kernel memory allocator. In *Proc. of the USENIX Summer Technical Conf.* (1994), p. 6.
- [21] CADAR, C., AND HOSEK, P. Multi-version software updates. In *Proc. of the Fourth Int'l Workshop on Hot Topics in Software Upgrades* (2012), pp. 36–40.
- [22] CHEN, H., CHEN, R., ZHANG, F., ZANG, B., AND YEW, P.-C. Live updating operating systems using virtualization. In *Proc. of the Second Int'l Conf. on Virtual Execution Environments* (2006), pp. 35–44.
- [23] CHEN, H., YU, J., CHEN, R., ZANG, B., AND YEW, P.-C. POLUS: A POverful live updating system. In *Proc. of the 29th Int'l Conf. on Software Eng.* (2007), pp. 271–281.
- [24] CHOW, J., PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proc. of the 14th USENIX Security Symp.* (2005), pp. 22–22.
- [25] CRAMERI, O., KNEZEVIC, N., KOSTIC, D., BIANCHINI, R., AND ZWAENEPOEL, W. Staged deployment in Mirage, an integrated software upgrade testing and distribution system. In *Proc. of the 21st ACM Symp. on Operating Systems Principles* (2007), pp. 221–236.
- [26] DÖBEL, B., HÄRTIG, H., AND ENGEL, M. Operating system support for redundant multithreading. In *Proc. of the 10th Int'l Conf. on Embedded software* (2012), pp. 83–92.
- [27] DHURJATI, D., AND ADVE, V. Backwards-compatible array bounds checking for C with very low overhead. In *Proc. of the 28th Int'l Conf. on Software Eng.* (2006), pp. 162–171.
- [28] DIMITROV, M., AND ZHOU, H. Unified architectural support for soft-error protection or software bug detection. In *Proc. of the 16th Int'l Conf. on Parallel Architecture and Compilation Techniques* (2007), pp. 73–82.
- [29] DUMITRAS, T., AND NARASIMHAN, P. Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system. In *Proc. of the 10th Int'l Conf. on Middleware* (2009), pp. 1–20.
- [30] DUMITRAS, T., NARASIMHAN, P., AND TILEVICH, E. To upgrade or not to upgrade: Impact of online upgrades across multiple administrative domains. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications* (2010), pp. 865–876.
- [31] ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., AND NOTKIN, D. Dynamically discovering likely program invariants to support program evolution. In *Proc. of the 21st Int'l Conf. on Software Eng.* (1999), pp. 213–224.
- [32] FONSECA, P., LI, C., AND RODRIGUES, R. Finding complex concurrency bugs in large multi-threaded applications. In *Proc. of the Sixth ACM European Conf. on Computer Systems* (2011), pp. 215–228.
- [33] GIUFFRIDA, C., CAVALLARO, L., AND TANENBAUM, A. S. Practical automated vulnerability monitoring using program state invariants. In *Proc. of the Int'l Conf. on Dependable Systems and Networks* (2013).
- [34] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proc. of the 21st USENIX Security Symp.* (2012), p. 40.
- [35] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Safe and automatic live update for operating systems. In *Proceedings of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (2013), pp. 279–292.
- [36] GIUFFRIDA, C., AND TANENBAUM, A. Safe and automated state transfer for secure and reliable live update. In *Proc. of the Fourth Int'l Workshop on Hot Topics in Software Upgrades* (2012), pp. 16–20.
- [37] GIUFFRIDA, C., AND TANENBAUM, A. S. Cooperative update: A new model for dependable live update. In *Proc. of the Second Int'l Workshop on Hot Topics in Software Upgrades* (2009), pp. 1–6.
- [38] GIUFFRIDA, C., AND TANENBAUM, A. S. A taxonomy of live updates. In *Proc. of the 16th ASCII Conf.* (2010).
- [39] GOODFELLOW, B. Patch tuesday. http://www.thetechgap.com/2005/01/strongpatch_tue.html.
- [40] GUPTA, D., AND JALOTE, P. On-line software version change using state transfer between processes. *Softw. Pract. and Exper.* 23, 9 (1993), 949–964.
- [41] GUPTA, D., JALOTE, P., AND BARUA, G. A formal framework for on-line software version change. *IEEE Trans. Softw. Eng.* 22, 2 (1996), 120–131.
- [42] HANGAL, S., AND LAM, M. S. Tracking down software bugs using automatic anomaly detection. In *Proc. of the 24th Int'l Conf. on Software Eng.* (2002), pp. 291–301.
- [43] HANSELL, S. Glitch makes teller machines take twice what they give. *The New York Times* (1994).
- [44] HAYDEN, C., SAUR, K., HICKS, M., AND FOSTER, J. A study of dynamic software update quiescence for multi-threaded programs. In *Proc. of the Fourth Int'l Workshop on Hot Topics in Software Upgrades* (2012), pp. 6–10.
- [45] HAYDEN, C., SMITH, E., HARDISTY, E., HICKS, M., AND FOSTER, J. Evaluating dynamic software update safety using systematic testing. *IEEE Trans. Softw. Eng.* 38, 6 (2012), 1340–1354.
- [46] HAYDEN, C. M., HARDISTY, E. A., HICKS, M., AND FOSTER, J. S. Efficient systematic testing for dynamically updatable software. In *Proc. of the Second Int'l Workshop on Hot Topics in Software Upgrades* (2009), pp. 1–5.
- [47] HAYDEN, C. M., MAGILL, S., HICKS, M., FOSTER, N., AND FOSTER, J. S. Specifying and verifying the correctness of dynamic software updates. In *Proc. of the Fourth Int'l Conf. on Verified Software: Theories, Tools, Experiments* (2012), pp. 278–293.

- [48] HAYDEN, C. M., SMITH, E. K., DENCHEV, M., HICKS, M., AND FOSTER, J. S. Kitsune: Efficient, general-purpose dynamic software updating for C. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications* (2012).
- [49] HAYDEN, C. M., SMITH, E. K., HICKS, M., AND FOSTER, J. S. State transfer for clear and efficient runtime updates. In *Proc. of the Third Int'l Workshop on Hot Topics in Software Upgrades* (2011), pp. 179–184.
- [50] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. Reorganizing UNIX for reliability. In *Proc. of the 11th Asia-Pacific Conf. on Advances in Computer Systems Architecture* (2006), pp. 81–94.
- [51] HOSEK, P., AND CADAR, C. Safe software updates via multi-version execution. In *Proc. of the Int'l Conf. on Software Engineering* (2013), pp. 612–621.
- [52] HÄRTIG, H., HOHMUTH, M., LIEDTKE, J., WOLTER, J., AND SCHÖNBERG, S. The performance of microkernel-based systems. In *Proc. of the 16th ACM Symp. on Oper. Systems Prin.* (1997), pp. 66–77.
- [53] HUNT, G. C., AND LARUS, J. R. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.* 41, 2 (2007), 37–49.
- [54] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *Proc. of the 22nd ACM Symp. on Oper. Systems Prin.* (2009), pp. 207–220.
- [55] KRAMER, J., AND MAGEE, J. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.* 16, 11 (1990), 1293–1306.
- [56] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the Int'l Symp. on Code Generation and Optimization* (2004), p. 75.
- [57] MAKRIKIS, K., AND BAZZI, R. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proc. of the USENIX Annual Tech. Conf.* (2009), pp. 397–410.
- [58] MAKRIKIS, K., AND RYU, K. D. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proc. of the Second ACM European Conf. on Computer Systems* (2007), pp. 327–340.
- [59] MAURER, M., AND BRUMLEY, D. TACHYON: Tandem execution for efficient live patch testing. In *Proc. of the 21st USENIX Security Symp.* (2012), p. 43.
- [60] NAGARAJA, K., OLIVEIRA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and dealing with operator mistakes in internet services. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation* (2004), pp. 5–5.
- [61] NEAMTIU, I., AND DUMITRAS, T. Cloud software upgrades: Challenges and opportunities. In *Proc. of the Int'l Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems* (2011), pp. 1–10.
- [62] NEAMTIU, I., AND HICKS, M. Safe and timely updates to multi-threaded programs. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation* (2009), pp. 13–24.
- [63] NEAMTIU, I., HICKS, M., FOSTER, J. S., AND PRATIKAKIS, P. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation* (2008), pp. 37–49.
- [64] NEAMTIU, I., HICKS, M., STOYLE, G., AND ORIOL, M. Practical dynamic software updating for C. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation* (2006), pp. 72–83.
- [65] NG, W. T., AND CHEN, P. M. The systematic improvement of fault tolerance in the Rio file cache. In *Proc. of the 29th Int'l Symp. on Fault-Tolerant Computing* (1999), p. 76.
- [66] OLIVEIRA, F., NAGARAJA, K., BACHWANI, R., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and validating database system administration. In *Proc. of the USENIX Annual Tech. Conf.* (2006), pp. 213–228.
- [67] O'REILLY, T. What is Web 2.0. <http://oreilly.com/pub/a/web2/archive/what-is-web-20.html>.
- [68] PATTABIRAMAN, K., SAGGESE, G. P., CHEN, D., KALBARCZYK, Z. T., AND IYER, R. K. Automated derivation of application-specific error detectors using dynamic analysis. *IEEE Trans. Dep. Secure Comput.* 8, 5 (2011), 640–655.
- [69] RESCORLA, E. Security holes... who cares? In *Proc. of the 12th USENIX Security Symp.* (2003), vol. 12, pp. 6–6.
- [70] ROWASE, O., AND LAM, M. S. A practical dynamic buffer overflow detector. In *Proc. of the 11th Annual Symp. on Network and Distr. System Security* (2004), pp. 159–169.
- [71] STOYLE, G., HICKS, M., BIERMAN, G., SEWELL, P., AND NEAMTIU, I. Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.* 29, 4 (2007).
- [72] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.* 23, 1 (2005), 77–110.
- [73] TUCEK, J., XIONG, W., AND ZHOU, Y. Efficient on-line validation with delta execution. In *Proc. of the 14th Int'l Conf. on Architectural support for programming languages and operating systems* (2009), pp. 193–204.
- [74] VANDEWOUDE, Y., EBRAERT, P., BERBERS, Y., AND D'HONDT, T. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Softw. Eng.* 33, 12 (2007), 856–868.
- [75] ZHENG, W., BIANCHINI, R., JANAKIRAMAN, G. J., SANTOS, J. R., AND TURNER, Y. JustRunIt: Experiment-based management of virtualized data centers. In *Proc. of the USENIX Annual Tech. Conf.* (2009), p. 18.

Live upgrading thousands of servers from an ancient Red Hat distribution to 10 year newer Debian based one.

Marc MERLIN
Google, Inc.

Abstract

Google maintains many servers and employs a file level sync method with applications running in a different partition than the base Linux distribution that boots the machine and interacts with hardware. This experience report first gives insights on how the distribution is setup, and then tackles the problem of doing a difficult upgrade from a Red Hat 7.1 image snapshot with layers of patches to a Debian Testing based distribution built from source. We will look at how this can actually be achieved as a live upgrade and without ending up with a long “flag day” where many machines are running totally different distributions, which would have made testing and debugging of applications disastrous during a long switchover period.

Like a coworker of mine put it, “It was basically akin to upgrading Red Hat 7.1 to Fedora Core 16, a totally unsupported and guaranteed to break upgrade, but also switching from rpm to dpkg in the process, and on live machines.”

The end of the paper summarizes how we designed our packaging system for the new distribution, as well as how we build each new full distribution image from scratch in a few minutes.

Tags: infrastructure, Linux, distribution, live upgrade

Introduction

The Linux operating system that Google uses in our service “production” environment has a strange history which will be described before explaining how we upgraded it.

Google’s production Linux OS is managed in three layers. The kernel and device drivers, user-space, and the running applications.

The kernel and device drivers are updated frequently and separately from the operating system. These files are maintained, fleet-wide, by a different team. Aside from obvious touch points, this maintenance is unrelated to the work described in this paper.

Each application runs in a chroot-ed jail. This jail includes all the programs, shared libraries, and data files required for the application to run. Therefore they are not entangled with the rest of the operating system. This independence from the underlying operating system is fairly extreme: even external libraries are statically linked. We provide multiple hermetic versions of python, the C++ libraries, the C runtime loader and libraries that applications can choose from. These are all decoupled from the booted operating system.

The remaining part is the user-space files - the init scripts, the /usr/bin binaries, and so on. The OS’s native package system is only used for this part, which is the focus of this paper.

Because of this decoupling the user-space portion could go a long time without upgrades. In fact, it remained at the equivalent of Red Hat 7.1 for many years.

Changing a fleet of thousands of machines from one distribution to another is a rare event and there is no “best practice” for doing so. One could convert small groups of machines until the entire fleet is converted. During the transition the fleet would contain two different operating systems. That was unacceptable - the entire Google fleet is kept within one or two minor OS revisions at any given time. Adding multiple operating systems would have multiplied complexity.

Instead we chose to transition parts of the OS one at a time: the boot scripts, the user-space binaries, the package system, etc. Over 3 years the entire OS would change, sometimes file by file, until it was completely replaced. This permitted each little step to be fully tested and possibly reverted. Most importantly users would not see a “flag day” change. At our a large scale, a small error is multiplied by thousands of machines. The ability to move slowly, cautiously, and with large amounts of testing, was critical.

System Administrators often refer to their work as “changing the tires while the car is driving down the highway”. In this case we changed the front left tire across the entire fleet. Once that was done we changed the steering wheel across the entire fleet. This process continued and after four years we had an entirely new car.

1. Google Servers and Linux, the early days

Like many startups, Google started with a Linux CD. It started around 1998 with a Red Hat 6.2 that was installed on the production machines. Soon thereafter, we got a kickstart network install, and it grew from there.

Updates and custom configurations were a problem. Machine owners had ssh loops to connect to machines and run custom install/update commands. At some point, they all got reinstalled with Red Hat 7.1 with custom software re-installed on top, but obviously this was not the right way to do things.

1.1. Better update management

The custom ssh loops were taking longer to run, and missing more machines each time. It was quick and dirty, but this has never scaled. Generally any push based method is doomed.

Now, it's not uncommon to run apt-get or yum from cron and hope updates will mostly work that way. However, for those of you who have tried running apt-get/dpkg/rpm/yum on thousands of servers, you may have found that random failures, database corruptions (for rpm) due to reboots/crashes during updates, and other issues make this not very reliable.

Even if the package DBs don't fail, it's often a pain to deal with updates to config files conflicting with packages, or unexpected machine state that breaks the package updates and causes all subsequent updates to fail until an admin fixes the machine manually, or a crude script simply wipes and re-installs the machine. The first method doesn't scale and the second one can cause data loss and outages.

1.2. Full file level filesystem sync

As crude as it is, file level syncing recovers from any state and bypasses package managers and their unexpected errors. It makes all your servers the same though, so custom packages and configs need to be outside of the synced area or manually excluded. Each server then has a list of custom files (network config, resolv.conf, syslog files, etc...) that are excluded from the sync.

Now, using rsync for entire machines off a master image doesn't scale well on the server side, and can bog the I/O on your clients, causing them to be too slow to serve requests with acceptable latency. You also need triggers that restart programs if certain files change.

So, we wrote custom rsync-like software where clients initiate file level syncs from a master image. It then al-

lows for shell triggers to be run appropriately. IO is throttled so that it does not negatively impact machines serving live requests while they are being upgraded.

1.3. Isolating server packages from the Server OS

We have custom per machine software that is outside of the centrally managed root partition, and therefore does not interfere with updates. In other words, the distribution is a fancy boot loader with housekeeping and hardware monitoring tools. Applications go in a separate partition and are not allowed to touch the dpkg/rpm database, or modify the root partition in any other way.

The software run by the server is typically run in a chroot with a limited view of the root partition, allowing the application to be hermetic and protected from root filesystem changes. We also have support for multiple libcs and use static linking for most library uses. This combination makes it easy to have hundreds of different apps with their own dependencies that change at their own pace without breaking if the OS that boots the machine changes.

The limited view of the root partition was achieved by first having a blacklist of what not to include in the chroot for applications, and later transitioning to a whitelist. In other words, our restricted chroot for user applications only contains files that have been opted in.

This upgrade itself also gave us a chance to find places where we weren't fully hermetic like we should have been.

2. How we did updates

Because had decoupled the booting OS from the applications running on top, the actual OS saw a minimal amount of updates. Updates were mostly security updates for bugs that did potentially affect us. From time to time we also needed a new feature that was added in the userland tools that we used. In other words OS updates were few and far in between and done only on demand. This is how we ended up still running something that was still mostly Red Hat 7.1 after about 10 years, managed by 2 or fewer people. In some ways, we pushed the "if it ain't broke, don't fix it" motto as far as we could.

2.1. Server image updates

We effectively had a filesystem image that got synced to a master machine, new packages were installed and the new image was snapshotted. We had scripts to store the new filesystem snapshot in Perforce, one of our

source control systems, and allow for crude diffing between the two images. The new golden image was then pushed to test machines, had to pass regression tests, and pushed to a test cluster, eventually with some live traffic. When the new image has seen enough testing, it is pushed slowly to the entire fleet.

2.2. Dealing with filesystem image updates

After dealing with the obvious issues of excluding machine specific config files, and logs from full filesystem syncs, the biggest difference is dealing with package postinstalls. We removed most of them since anything that is meant to run differently on each machine doesn't work with a golden image that is file-synced.

Examples:

- Running `ldconfig` after a library change is ok.
- Creating files in postinstall works, but is undesirable since those don't show up in the package file list.
- For the case of files like `ssh` host keys, it's obviously bad to create a single host key that gets snapshotted and synced everywhere.
- Re-running `lilo` after updating `lilo.conf` would not work.
- Restarting daemons doesn't work either.
- Many postinstalls have code to deal with cleaning up for upgrades that weren't relevant to us, so they could be ignored or removed.

We dealt with postinstalls that were necessary on a case by case basis and we used our filesystem sync post push triggers that restart daemons or re-install `lilo` boot blocks after the relevant config files or binaries were updated.

2.3. Testing software before doing image updates

We wrote a `test-rpm-install/test-deb-install` script that takes a clean machine, installs the package, and gets a before/after snapshot of the entire filesystem. This allowed us to verify what gets added/removed to the filesystem, as well as review unix permission changes, and size increases. We always fought software bloat, which is how we managed to keep a small boot image after years of evolution (it actually shrunk in size over time as bloat was identified and removed).

Software engineers of course have mandated code reviews and unit tests for their software. Once those are done for a change, we build an image with just the new package and send it to our regression tester. The regression tester runs on a sample of our different platforms, applies the update without rebooting, and ensures that critical daemons and services continue to work after

the update. Once that works, the machine is rebooted, the services checked again, after which the machine is rebooted first cleanly, and then a second time as a crash reboot. If this all passes, the image is then reverted, we make sure daemons do not misbehave when downgraded (this can happen if the old code cannot deal with state files from the new code), and the downgraded image is then rebooted to make sure everything comes back up as expected.

While this test suite is not foolproof, it has found a fair amount of bugs, and ideally let the software submitter find problems before submitting the package for inclusion in the next image cut.

2.4. Reviewing image updates before deployment, and test deployment.

We start with the old image's files checked into Perforce (Perforce was mostly chosen because it was our main already in use source control system at the time). Metadata was stored into a separate file that wasn't much reviewable (dev nodes, hardlinks, permissions, etc...), but we had a reviewer friendly `ls -alR` type file list to review permission and owner changes.

Image build input was a list of pre-approved packages to update with package owners providing their own testing notes, and features they're looking at adding. They got installed on a test machine, and the output was a new filesystem image where Perforce allowed reviewing diffs of ASCII files, and we could review changes in binary sizes as well as file permissions. From there, if approved, the image was sent to a pool of early test machines, and deployed slowly fleet-wide if no one complained about regressions.

2.5. Admin and debugging considerations

While the focus of this paper is on distribution management and upgrades, there are a few things worth noting related to management of headless machines and debugging boot issues. We have serial consoles on some test machines, but it's not worth the price on all machines. As a result we use `bootlogd` to capture boot messages without requiring the much heavier and buggy `plymouth`. We also start a debug `sshd` before the root filesystem is `fsck`'ed and remounted read-write. That way we can easily probe/debug machines that aren't rebooting properly or failing to `fsck` their root filesystem.

When you have so many machines, you want to keep the init system simple and dependable. Whenever possible we want all our machines to behave the same. As a result, we stuck with normal `init`, and looked at Debian's `insserv` and `startpar` for simple dependency boot-

ing that we can set in stone and review at image creation time. Both upstart and systemd require way too many moving pieces and introduce boot complexity and unpredictability that was not worth the extra boot time they could save.

While shaving a few seconds of boot isn't that important to us, we do save reboot time by avoiding double reboots when the root filesystem needs repair, and do so by doing a pivot-root to an initramfs with busybox, release the root filesystem, fsck it, and then pivot-root back to it to continue normal boot.

2.6 This worked amazingly well over time, but it had many issues

- Like is often the case, our system was not carefully thought out and designed from the ground up, but just a series of incremental “we have to fix this now” solutions that were the best the engineers with limited time could do at the time.
- Our entire distribution was really just a lot of overlaid patches on top of a Red Hat 7.1 live server snapshot almost 10 years ago.
- A lot of software was still original Red Hat 7.1 and we had no good way to rebuild it on a modern system. Worse, we just assumed that the binaries we had were indeed built from the original Red Hat source.
- The core of our distribution was now very old, and we knew we couldn't postpone upgrading it forever, but had no good plan for doing so.

3.0. Upgrade Plan

3.1 Which distribution?

Back in the days, we wasted too much time building open source software as rpms, when they were available as debs. As a result, we were not very attached to Red Hat due to the lack of software available in rpm form vs what was available in Debian. We had already switched away from Red Hat on our Linux workstations years prior for the same reason (our workstations are running a separately maintained linux distribution because they have different requirements and tradeoffs than our servers do) Back then, Red Hat 9 had 1,500 packages vs 15,000 in Debian. Today Fedora Core 18 has 13,500 vs 40,000 in Debian testing. Arguably Red Hat fares better today than it did then, but still remains inferior in software selection.

As a result, ProdNG, our Linux distribution built from source, was originally based off Ubuntu Dapper. At the time Ubuntu was chosen because we were also using it

on our workstations. Later on, we switched to straight Debian due to Ubuntu introducing several forced complexities that were not optional and not reliable when they were introduced, like upstart and plymouth.

3.2 ProdNG Design

Richard Gooch and Roman Mitnitski, who did the original design for the new distribution came up with these design points to address the limitations of our existing distribution:

- Self hosting.
- Entirely rebuilt from source.
- All packages stripped of unnecessary dependencies and libraries (xml2, selinux library, libacl2, etc..)
- Less is more: the end distribution is around 150MB (without our custom bits). Smaller is quicker to sync, re-install, and fsck.
- No complicated upstart, dbus, plymouth, etc. Tried and true wins over new, fancy and more complex, unless there is measurable benefit from the more complex version.
- Newer packages are not always better. Sometimes old is good, but only stay behind and fork if really necessary. On the flip side, do not blindly upgrade just because upstream did.
- Hermetic: we create a ProdNG chroot on the fly and install build tools each time for each new package build.
- Each image update is built by reassembling the entire distribution from scratch in a chroot. This means there are no upgrades as far as the package management is concerned, and no layers of patches on top of a filesystem that could have left-over forgotten cruft.

3.3 Upgrade Plan

Once we had a ProdNG distribution prototype that was booting, self-hosting, and ready to be tested, we all realized that switching over would be much harder than planned.

There was no way we could just roll out a brand new distribution that was 100% different from the old one, with software that was up to 10 years newer, and hope for the best. On top of that, our distribution contains our custom software that is required for new hardware bringup, or network changes, so we could not just have paused updates to the old distribution for months while we very slowly rolled out the new one. Cycles of find a bug, pause the rollout or revert, fix the bug (either in the distribution, or in the software that relied on the behavior of the old one), and try again, could have potentially lasted for months.

It could have been possible with a second team to maintain the old production image in parallel and at each review cycle build 2 distributions, but this had many problems. To list just a few:

- Double the review load, but it was obviously not desirable, nor really achievable with limited staffing.
- Would we really want to have a non-uniform setup in production for that long? That's not going to make debugging easy in case we notice failures in production and for months we'd now first have to worry about whether "Is it a ProdNG related problem, or a distribution independent problem?". Our monitoring tools expect the same distribution everywhere, and weren't designed to quickly categorize errors based on ProdNG or not ProdNG. This could have been done with a lot of work, but wasn't deemed a good use of time when there was a better alternative (explained below).
- With one distribution made with rpms, while the other one is dpkg, using totally different build rules and inter package dependencies, our package owners would also have a lot more work.
- While it's true that we have few internal users who depend on the distribution bits, that small number, from people working on the installers, and people writing software managing machine monitoring, hardware, and software deployment are still a sizeable amount of people (more than just a handful we can sync with or help individually if we change/break too many things all at once).

One motto at Google is that one team should not create a lot of work for other teams to further their own agenda, unless it's absolutely unavoidable and the end goal is worth it. At the time, we were not able to make a good enough case about the risk and work we would have introduced. In hindsight, it was a good call, the switch if done all at once, would have introduced way too many problems that were manageable handled one by one over time, but not as much if thrown around all the same time.

Around that time, Roman had to go back to another project, and with no good way to push ProdNG forward due to the risk of such a big change, and impact on other internal teams, it stalled.

3.4 The seemingly crazy idea that worked

Later, at the time I joined the team working on the production image, Richard Gooch and I sat down to list the requirements for a successful upgrade:

- We need to keep all the machines in a consistent state, and only stay with 2 images: the current/old one and the new one being pushed.

- If flag day there must be, it must be as short a day as possible.
- Service owners should not notice the change, nor should their services go down.
- rpm vs dpkg should be a big switch for us, the maintainers, but not the server users.
- There are just too many changes, from coreutils to others, for the jump to be small enough to be safe.
- And since we can't have a big jump, we can't jump at all.

Richard came up with the idea of slowly feeding our ProdNG distribution into our existing production image, a few packages at a time during each release cycle. Yes, that did mean feeding deps into an rpm distro.

To most, it likely sounded like a crazy idea because it was basically akin to upgrading Red Hat 7.1 to Fedora Core 16, a totally unsupported and guaranteed to break upgrade, but also switching from rpm to dpkg in the process, and on live machines.

An additional factor that made this idea "crazy" is that our ProdNG packages were based on libc 2.3.6 whereas our production image was based on libc 2.2.2, thus ProdNG binaries would simply not run on the old image, and it was unsafe to upgrade the system libc without recompiling some amount of its users. Richard had a key insight and realized that binary patching the ProdNG binaries would allow them to run on the old image. Since the original ProdNG prototype was developed and shelved, the production image had acquired a hermetic C library for the use of applications outside of the OS (this allowed applications to be use a libc, and later among several available, without relying on the one from the OS).

At the time, his hermetic C library was also based on libc 2.3.6 and thus ProdNG binaries could use it as long as the run-time linker path in the ELF header was binary patched with a pathname of the same length.

Since doing unsupported live upgrades has been a side hobby of mine since Red Hat 2.1, including switching binary formats from zmagic to qmagic (libc4), then ELF with libc5, and finally glibc with libc6, I didn't know how long it would take, but I figured this couldn't be any worse and that I could make it happen.

3.5 Implementing the slow upgrade

By then, ProdNG was still self hosting, and could build new packages, so Richard wrote an alien(1) like package converter that took a built ProdNG package and converted it to an rpm that would install on our current production image (this did include some sed hackery to

convert dependency names since Debian and Red Hat use different package names for base packages and libraries that are required for other packages to install), but overall it was not that complicated. The converter then ran the binary patcher described above, and ran an ugly script I wrote to turn Debian changelogs into Red Hat ones so that package upgrades would show expected changelog diffs for the reviewers.

Because by the time I joined to help the production image group, the ProdNG build had been stale for a couple of years, I started by refreshing ProdNG, and package by package, upgrading to more recent source if applicable, stripping all the new features or binaries we didn't need, and feeding the resulting package as a normal rpm package upgrade in the next production image release.

From there, I looked at our existing binaries, and checked whether they would just work if libc was upgraded and they weren't recompiled. Most passed the test without problem, while a few showed

```
Symbol `sys_siglist' has different size in
shared object, consider re-linking
```

The other issue was that some binaries were statically linked, and those have hardcoded pathnames to libnss libraries, which were the ones we were trying to remove. Having non matching libc and libnss also caused those binaries to fail, which wasn't unexpected. This problem was however quickly solved by removing the old libc altogether and repointing ld-linux.so to the new libc. I then added a few symlinks between the location of the libnss libs from the old libc to the ones from the new libc.

Note that we had to run with this dual libc configuration for a while since we still had a self imposed rule of only upgrading a few packages at each cycle. Therefore we pushed fixed packages a few at a time until we were ready one day to remove the old libc and replace it with symlinks to the new one.

3.6 If you can delete it, you don't have to upgrade it

Despite of the fact that our image was a snapshot of a live Red Hat 7.1 server install, it contained a lot of packages that didn't belong in a base server install, or packages that we didn't need for our uses.

Distributions with crazy dependency chains have only been getting worse over time, but even in the Red Hat 7.1 days, dependencies in Red Hat were already far from minimal. Some were pure cruft we never needed (X server, fonts, font server for headless machines without X local or remote, etc...). Next, I looked for all

things that made sense to ship as part of RH 7.1, but were useless to us (locales and man pages in other languages, i18n/charmaps, keyboard mappings, etc...).

After that, I looked for the next low hanging fruit and found libraries nothing was using anymore (left over from prior upgrade, or shipped by default, but not used by us). For some libraries, like libwrap, I was able to remove them after upgrading the few packages that used them, while omitting the library from their builds.

When it was all said and done, I had removed 2/3rd of the files we had in the initial image, and shed about 50% of the disk space used by the Linux image (not counting our custom in-house software).

3.7 The rest of the upgrade

What didn't get deleted, had to be upgraded however. Once the libc hurdle was past, it was a lot of painstaking work to deal with each weird upgrade differently, and qualify each big software jump for things like cron, or syslog, to be sure they would be safe and not fix a bug that we were relying on. Just upgrading rsync from 2.x to 3.x took 4 months of work because of semantics that changed in the code in how it handled permission syncs, and our dependence on the old behavior.

Our distribution was so old that it didn't even have coreutils. It had fileutils + textutils + sh-utils, which got replaced with fairly different binaries that unfortunately were not backward compatible so as to be more POSIX compliant. Upgrading just that took a lot of effort to scan all our code for instances of tail +1, or things scanning the output of ls -l. In the process, multiple utilities got moved from /bin to /usr/bin, or back, which broke some scripts that unfortunately had hardcoded paths.

Aside from a couple of upgrades like coreutils, there weren't too many upgrades with crazy dependency chains, so it was not a problem to upgrade packages a few at a time (5 to 10 max each time).

On some days, it was the little things. The day I removed /etc/redhat-release, it broke a bunch of java code that parsed this file to do custom things with fonts depending on the presence of that file. At Google, whoever touched something last is responsible for the breakage, even if the bug wasn't in that change, so that typically meant that I had to revert the change, get the right team to fix the bug, wait for them to deploy the fix on their side, and then try again later.

3.8 Dealing with left over junk

Because our original image was a full filesystem image that got snapshotted in Perforce, we ended up with files that were not runtime created, and not part of any package either. We had junk that we didn't always know the source of, or sometimes whether it was safe to remove.

I ended up finding the expected leftover files (.rpm-save, old unused files), lockfiles and logfiles that shouldn't have been checked in and /etc/rcxx initscript symlinks. Any actual program that wasn't part of a package, was identified and moved to a package.

Then, I had to scan the entire filesystem for files that were not in a package, work through what was left on the list and deal with the entries on an case by case basis.

That said, the goal was never to purge every single last trace of Red Hat. We have some Red Hat pathnames or functions left over to be compatible with things that expect Red Hat and aren't quite LSB compliant. We only removed Red Hat specific bits (like rpm itself) when it was simple to do so, or because maintaining them long term was going to be more work than the cost of removal.

3.9 A difficult problem with /etc/rc.d/...

Back in the day (mid 1990's) someone at Red Hat misread the linux filesystem standard and put the initscripts in /etc/rc.d/rc[0-6].d and /etc/rc.d/nit.d instead of /etc/rc[0-6].d, as implemented in other linux distributions including Debian. Migrating to Debian therefore included moving from /etc/rc.d/init.d to /etc/init.d.

Unfortunately I found a bug in our syncing program when switching from /etc/rc.d/init.d (Red Hat) to /etc/init.d (Debian): when the image syncer applied a new image that had /etc/init.d as the directory and /etc/rc.d/init.d as the compatibility symlink, that part worked fine, but then it also remembered that /etc/rc.d/init.d was a directory in the old image that got removed, and by the time it did a recursive delete of /etc/rc.d/init.d, it followed the /etc/rc.d/init.d symlink it had just created and proceeded to delete all of /etc/init.d/ it also had just created.

The next file sync would notice the problem and fix it, but this left machines in an unbootable state if they were rebooted in that time interval and this was not an acceptable risk for us (also the first file sync would trigger restarts of daemons that had changed, and since the initscripts were gone, those restarts would fail).

This was a vexing bug that would take a long time to fix for another team who had more urgent bugs to fix

and features to implement. To be fair, it was a corner case that no one had ever hit, and no one has hit since then.

This was a big deal because I had to revert the migration to /etc/init.d, and some of my coworkers pushed for modifying Debian forever to use /etc/rc.d/init.d. Putting aside that it was a bad hack for a software bug that was our fault, it would have been a fair amount of work to modify all of Debian to use the non standard location, and it would have been ongoing work forever for my coworkers after me to keep doing so. I also knew that the changes to initscripts in Debian would force us to have local patches that would cause subsequent upstream changes to conflict with us, and require manual merges.

So, I thought hard about how to work around it, and I achieved that by keeping Debian packages built to use /etc/init.d, but by actually having the real filesystem directory be /etc/rc.d/init.d while keeping /etc/init.d as a symlink for the time being. This was done by setting those up before Debian packages were installed in our image builder. Dpkg would then install its files in /etc/init.d, but unknowing follow the symlink and install them in /etc/rc.d/init.d.

This was ok, but not great though because we'd have a mismatch between the Debian file database and where the files really were on disk, so I worked further to remove /etc/rc.d/init.d.

We spent multiple months finding all references to /etc/rc.d/init.d, and repoint them to /etc/init.d. Once this was finished, we were able to remove the image build hack that created /etc/rc.d/init.d.

The bug did not trigger anymore because our new image did not have a /etc/rc.d/init.d compatibility symlink, so when the file syncer deleted the /etc/rc.d/init.d directory, all was well.

3.10 Tracking progress

Our converted ProdNG packages had a special extension when they were converted to RPMs, so it was trivial to use rpm -qa, look at the package names and see which ones were still original RPMs and which ones were converted debs.

I then used a simple spreadsheet to keep track of which conversions I was planning on doing next, and for those needing help from coworkers who had done custom modifications to the RPMs, they got advance notice to port those to a newer Debian package, and I worked with them to make a ProdNG package to upgrade their old RPM. They were then able to monitor the upgrade of their package, and apply the relevant tests to ensure that the package still did what they

needed. This allowed porting our custom patches and ensuring that custom packages were upgraded carefully and tested for their custom functionality before being deployed (we do send out patches upstream when we can, but not all can be accepted).

3.11 Communication with our internal users

We used different kinds of internal mailing lists to warn the relevant users of the changes we were about to make. Some of those users were the ones working on the root partition software, others were our users running all the software that runs google services inside the chroots we provide for them, and we also warned the people who watch over all the machines and service health when we felt we were making changes that were worth mentioning to them.

All that said, many of those users also had access to our release notes and announcements when we pushed a new image, and quickly knew how to get image diffs when debugging to see if we made an image change that might have something to do with a problem they are debugging.

4.0 Getting close to swichover time

After almost 3 years of effort (albeit part time since I was also working on maintaining and improving the current rpm based image, working with our package owners, as well as shepherding releases that continued to go out in parallel), the time came when everything had been upgraded outside of /sbin/init and Red Hat initscripts.

Checking and sometimes modifying Debian initscripts to ensure that they produced the same behavior that we were getting from our Red Hat ones took careful work, but in the end we got ProdNG to boot and provide the same environment as our old Red Hat based image. To make the migration easier, I fed shell functions from Red Hat's /etc/init.d/functions into Debian's /lib/lsb/init-functions and symlinked that one to /etc/init.d/functions. This allowed both Red Hat and Debian initscripts from 3rd party packages to just work.

4.1 Reverse conversions: rpms to debs

By then, a portion of our internal packages had been converted from rpms to debs, but not all had been, so we used reverse converter that takes rpms, and converts them to debs, with help from alien. The more tedious part was the converter I wrote to turn mostly free form Red Hat changelogs into Debian changelogs which

have very structured syntax (for instance, rpms do not even require stating which version of the package a changelog entry was for, and if you do list the version number, it does not check that the latest changelog entry matches the version number of the package). Rpm changelogs also do not contain time of day, or time-zones (I guess it was Raleigh-Durham Universal Time), so I just had to make those up, and problems happen if two rpm releases happened on the same day with no time since it creates a duplicate timestamp in the debian changelog. Some fudging and kludges were required to fix a few of those.

4.2 Time to switch

By then, ProdNG was being built in parallel with the rpm production image and they were identical outside of initscripts, and rpm vs dpkg. With some scripting I made the ProdNG image look like a patch image upgrade for the old image, and got a diff between the two. We did manual review of the differences left between 2 images (file by file diff of still 1000+ files). There were a few small differences in permissions, but otherwise nothing that wasn't initscripts or rpm vs dpkg database info.

It then became time to upgrade some early test machines to ProdNG, make sure it did look just like the older image to our internal users, and especially ensure that it didn't have some bugs that only happened on reboot 0.5% of the time on just one of our platforms. Then, it started going out to our entire fleet, and we stood around ready for complaints and alerts.

4.3 Switch aftermath

Early deployment reports found one custom daemon that was still storing too much data in /var/run. In Red Hat 7.1, /var/run was part of the root filesystem, while in ProdNG it was a small tmpfs. The daemon was rebuilt to store data outside of /var/run (we have custom locations for daemons to write bigger files so that we can control their sizes and assign quotas as needed, but this one wasn't following the rules).

Most of the time was actually spent helping our package owners convert their rpms to debs and switching to new upload and review mechanisms that came with ProdNG since the image generation and therefore review were entirely different.

As crazy as the project sounded when it started, and while it took awhile to happen, it did. Things worked out beautifully considering the original ambition.

4.4 Misc bits: foregoing dual package system support

We had code to install rpms unmodified in our ProdNG deb image, and even have them update the dpkg file list. We however opted for not keeping that complexity since dual package support would have rough edges and unfortunate side effects. We also wanted to entice our internal developers to just switch to a single system to make things simpler: debs for all. They are still able to make rpms if they wish, but they are responsible for converting them to debs before providing them to us.

As a side result, we were able to drop another 4MB or so of packages just for just rpm2cpio since rpm2cpio required 3-4MB of dependencies. I was able to find a 20 line shell script replacement on the net that did the job for us. This allowed someone to unpack an old legacy rpm if needed while allowing me to purge all of rpm and its many libraries from our systems.

Debian made a better choice by having an archive system that can be trivially unpacked with ar(1) and tar(1) vs RPM that requires rpm2cpio (including too many rpm libraries) and still loses some permissions which are saved as an overlay stored inside the RPM header and lost during rpm2cpio unpacking.

4.5 No reboots, really?

I stated earlier that the upgrades we pushed did not require to reboot servers. Most services could just be restarted when they got upgraded without requiring a reboot of the machine.

There were virtually no times where we had code that couldn't be re-exec'ed without rebooting (even /sbin/init can re-exec itself), that said our servers do get occasionally rebooted for kernel upgrades done by another team, and we did benefit from those indirectly for cleaning up anything in memory, and processed that didn't restart, if we missed anyway.

5.0 ProdNG Design Notes

While it's not directly related to the upgrade procedure, I'll explain quickly how the new image is designed.

5.1 ProdNG package generation

Since one of the goals of our new distribution was to be self-hosting, and hermetic, including building a 32bit multiarch distribution (32bit by default, but with some 64bit binaries), it made sense to build ProdNG packages within a ProdNG image itself. This is done by quickly unpacking a list of packages provided in a dependencies file (mix of basic packages for all builds,

and extra dependencies you'd like to import in that image to build each specific package). Debian provides pbuilder which also achieves that goal, but our method of unpacking the system without using dpkg is much faster (1-2 minutes at most), so we prefer it.

We use the debian source with modifications to debian/rules to recompile with fewer options and/or exclude sub-packages we don't need. We then have a few shell scripts that install that unpacked source into a freshly built ProdNG image, build the package, and retrieve/store the output. You get flexibility in building a package in an image where for instance libncurses is not available and visible to configure, while being present in the image currently deployed (useful if you'd like to remove a library and start rebuilding packages without it).

After package build, we have a special filter to prune things we want to remove from all packages (info pages, man pages in other languages, etc...) without having to modify the build of each and every package to remove those. The last step is comparing the built package against the previous package, and if files are identical, but the mtime was updated, we revert the mtime to minimize image review diffs later.

5.2 ProdNG image generation

This is how we build our images in a nutshell: each new image to push is generated from scratch using the latest qualified packages we want to include into it (around 150 base Linux packages).

The image is built by retrieving the selected packages, unpacking them in a chroot (using ar and untar), and chrooting into that new directory. From there, the image is good enough to allow running dpkg and all its dependencies, so we re-install the packages using dpkg, which ensures that the dpkg database is properly seeded, and the few required postinstall scripts do run. There are other ways to achieve this result (debootstrap), but because our method runs in fewer than 10 minutes for us and it works, we've stuck with it so far.

As explained, package builds revert mtime only changes, and squash binary changes due to dates (like gzip of the same man page gives a new binary each time because gzip encodes the time in the .gz file). We have a similar patch for .pyc files. As a result of those efforts, rebuilding an image with the same input packages is reproducible and gives the same output.

5.3 ProdNG image reviews

The new ProdNG images are not checked in Perforce file by file. We get a full image tar.gz that is handed off to our pusher and reviews are done by having a script unpack 2 image tars, and generate reviewable reports for it:

- file changes (similar ls -alR type output)
- which packages got added/removed/updated
- changelog diffs for upgraded packages
- All ASCII files are checked into Perforce simply so that we can track their changes with Perforce review tools.
- compressed ASCII files (like man pages or docs) are uncompressed to allow for easy reviews.
- Other binary files can be processed by a plugin that turns them into reviewable ASCII.

6. Lessons learned or confirmed.

1. If you have the expertise and many machines, maintaining your own sub Linux distribution in house gives you much more control.
2. At large scales, forcing server users to use an API you provide, and not to write on the root FS definitely helps with maintenance.
3. File level syncing recovers from any state and is more reliable than other methods while allowing for complex upgrades like the one we did.
4. Don't blindly trust and install upstream updates. They are not all good. They could conflict with your config files, or even be trojaned.
5. If you can, prune/remove all services/libraries you don't really need. Fewer things to update, and fewer security bugs to worry about.
6. Upgrading to the latest Fedora Core or Ubuntu from 6 months ago is often much more trouble than it's worth. Pick and chose what is worthwhile to upgrade. Consider partial upgrades in smaller bits depending on your use case and if your distribution is flexible enough to allow them.
7. Prefer a distribution where you are in control of what you upgrade and doesn't force you into an all or nothing situation. Ubuntu would be an example to avoid if you want upgrade flexibility since it directly breaks updates if you jump intermediate releases. Debian however offers a lot more leeway in upgrade timing and scope.
8. Keep your system simple. Remove everything you know you don't need. Only consider using upstart or systemd if you really know how their internals, pos-

sible race conditions, and are comfortable debugging a system that fails to boot.

References

As you may imagine, we didn't really have much existing work we were able to draw from. Alien(1) from Joey Hess definitely helped us out for the rpm to deb conversions, and here is the URL to the rpm2cpio I found to replace our 4MB of binaries:

<https://trac.macports.org/attachment/ticket/33444/rpm2cpio>

Acknowledgements

I need to single out my coworkers who helped design ProdNG: Richard Gooch and Roman Mitnitski for designing and implementing the first ProdNG images built from source and self-hosting. Richard Gooch also gets the credit for coming up with the initial design of feeding ProdNG in our existing file-synced image package by package by converting ProdNG debbs to rpms, as well as reviewing this paper.

Joey Hess, Debian maintainer of debhelpers and many other packages related to dpkg, is also the maintainer of alien(1). He was helpful on multiple occasions with dpkg/debhelper related questions when I was doing complicated conversions. Joeh also helped me include a cleaner version of a patch I had to write to properly convert some unix ownership and permissions from rpm to deb that alien(1) was failing to convert at the time.

Next, I would like to thank Tom Limoncelli for reviewing this paper as well as encouraging me to write it in the first place. He also gets credit for contributing the introduction which he was able to write with an outsider's point of view.

Finally, I owe thanks to the LISA reviewers for their review comments, and specifically Andrew Lusk for a careful and thorough final review of this paper.

Of course, I also need to thank all the coworkers at Google who helped me in various ways with this project, or doing our day to day maintenance and release work.

Availability

This document is available at the USENIX Web site.

Managing Smartphone Testbeds with SmartLab

Georgios Larkou
Dept. of Computer Science
University of Cyprus
glarkou@cs.ucy.ac.cy

Constantinos Costa
Dept. of Computer Science
University of Cyprus
costa.c@cs.ucy.ac.cy

Panayiotis G. Andreou
Dept. of Computer Science
University of Cyprus
panic@cs.ucy.ac.cy

Andreas Konstantinidis
Dept. of Computer Science
University of Cyprus
akonstan@cs.ucy.ac.cy

Demetrios Zeinalipour-Yazti
Dept. of Computer Science
University of Cyprus
dzeina@cs.ucy.ac.cy

Abstract

The explosive number of smartphones with ever growing sensing and computing capabilities have brought a paradigm shift to many traditional domains of the computing field. Re-programming smartphones and instrumenting them for application testing and data gathering at scale is currently a tedious and time-consuming process that poses significant logistical challenges. In this paper, we make three major contributions: First, we propose a comprehensive architecture, coined SmartLab¹, for managing a cluster of both real and virtual smartphones that are either wired to a private cloud or connected over a wireless link. Second, we propose and describe a number of Android management optimizations (e.g., command pipelining, screen-capturing, file management), which can be useful to the community for building similar functionality into their systems. Third, we conduct extensive experiments and microbenchmarks to support our design choices providing qualitative evidence on the expected performance of each module comprising our architecture. This paper also overviews experiences of using SmartLab in a research-oriented setting and also ongoing and future development efforts.

1 Introduction

Last year marked the beginning of the post PC era², as the number of smartphones exceeded for the first time in history the number of all types of Personal Computers (PCs) combined (i.e., Notebooks, Tablets, Netbooks and Desktops). According to IDC³, Android is projected to dominate the future of the smartphone industry with a share exceeding 53% of all devices shipped in 2016. Currently, an Android smartphone provides access to more than 650,000 applications, which bring unprecedented possibilities, knowledge and power to users.

Re-programming smartphones and instrumenting them for application testing and data gathering at scale is currently a tedious, time-consuming process that poses significant logistical challenges. To this end, we have implemented and demonstrated *SmartLab* [21], a comprehensive architecture for managing a cluster of both *Android Real Devices (ARDs)* and *Android Virtual Devices (AVDs)*, which are managed via an intuitive web-based interface. Our current architecture is ideal for scenarios that require *fine-grained* and *low-level* control over real smartphones, e.g., OS, Networking, DB and storage [20], security [5], peer-to-peer protocols [22], but also for scenarios that require the engagement of physical sensors and geo-location scenarios [38],[24]. Our preliminary release has been utilized extensively in-house for our research and teaching activities, as those will be overviewed in Section 7.

SmartLab's current hardware consists of over 40 Android devices that are connected through a variety of means (i.e., *wired*, *wireless* and *virtual*) to our *private cloud (datacenter)*, as illustrated in Figure 1. Through an intuitive web-based interface, users can upload and install Android executables on a number of devices concurrently, capture their screen, transfer files, issue UNIX shell commands, "feed" the devices with GPS/sensor mockups and many other exciting features. In this work, we present the anatomy of the SmartLab Architecture, justifying our design choices via a rigorous micro-benchmarking process. Our findings have helped us enormously in improving the performance and robustness of our testbed leading to a new release in the coming months.

Looking at the latest trends, we observe that open smartphone OSs, like Android, are the foundation of emerging Personal Gadgets (PGs): eReaders (e.g., Barnes & Noble), Smartwatches (e.g., Motorola MO-TOACTV), Raspberry PIs, SmartTVs and SmartHome appliances in general. SmartLab can be used to allow users manage all of their PGs at a fine-grain granular-

¹Available at: <http://smartlab.cs.ucy.ac.cy/>

²Feb. 3, 2012: Canalys, <http://goo.gl/T81iE>

³Jul. 6, 2012: IDC Corp., <http://goo.gl/CtDAC>



Figure 1: **Subset of the SmartLab smartphone fleet connected locally to our datacenter. More devices are connected over the wireless and wired network.**

ity (e.g., screen-capture, interactivity, filesystem). Additionally, we anticipate that the overtake of PC sales by Smartphone sales will soon also introduce the notion of Beowulf-like or Hadoop-like smartphone clusters for power-efficient computations and data analytics.

Moreover, one might easily build powerful computing testbeds out of deprecated smartphones, like *Microcellstores* [16], as users tend to change their smartphones more frequently than their PC. Consequently, providing a readily available PG management middleware like SmartLab will be instrumental in facilitating these directions. Finally, SmartLab is a powerful tool for Internet service providers and other authorities that require to provide remote support for their customers as it can be used to remotely control and maintain these devices. The contributions of this work are summarized as follows:

- i) **Architecture:** We present the architecture behind SmartLab, a first-of-a-kind open smartphone programming cloud that enables fine-grained control over both ARDs and AVDs via an intuitive web-based interface;
- ii) **Microbenchmarks:** We carry out an extensive array of microbenchmarks in order to justify our implementation choices. Our conclusions can be instrumental in building more robust Android smartphone management software in the future;
- iii) **Experiences:** We present our research experiences from using SmartLab in four different scenarios including: trajectory benchmarking [38], peer-to-peer benchmarking [22], indoor localization testing [24] and database benchmarking; and
- iv) **Challenges:** We overview ongoing and future developments ranging from Web 2.0 extensions to urban-scale deployment and security studies.

The rest of the paper is organized as follows: Section 2 looks at the related work, Section 3 presents our SmartLab architecture, while subsequent sections focus on the individual subsystems of this architecture: Section 4 covers power and connectivity issues, Section 5 provides a rigorous analysis of the *Android Debug Bridge (ADB)* used by our SmartLab *Device Server (DS)* presented in Section 6. Section 7 summarizes our research and teaching activities using SmartLab, Section 8 enumerates our ongoing and future developments while Section 9 concludes the paper.

2 Related Work

This section provides a concise overview of the related work. SmartLab has been inspired by *PlanetLab* [30] and *Emulab* [17], both of which have pioneered global research networks; *MoteLab* [37], which has pioneered sensor network research and *Amazon Elastic Compute Cloud (EC2)*. None of the aforementioned efforts focused on smartphones and thus those testbeds had fundamentally different architectures and desiderata. In the following subsections, we will overview testbeds that are related to SmartLab.

2.1 Remote Monitoring Solutions

There are currently a variety of Remote Monitoring Solutions (RMSs), including *Nagios* [26], a leading open-source RMS for over a decade, the Akamai Query System [9], STORM [14] and RedAlert [34]. All of these systems are mainly geared towards providing solutions for web-oriented services and servers. Moreover, none of those RMSs provide any tools related to the configuration and management of smartphone clusters. SmartLab focuses on providing a remote monitoring solution specifically for a smartphone-oriented cloud.

2.2 Wireless Sensor Network Testbeds

MoteLab [37] is a Web-based sensor network testbed deployed at Harvard University that has pioneered sensor network research. *CitySense* [27] has been MoteLab's successor enabling city-scale sensor network deployments. *Mobiscope* [1] is a federation of distributed mobile sensors into a taskable sensing system that achieves high density sampling coverage over a wide area through mobility. EU's *WISEBED* project [11] also federated different types of wireless sensor networks. Microsoft has made several attempts in building Sensor Networks with mobile phones [18], but none of these efforts has focused on smartphones in particular and their intrinsic characteristics like screen capturing, interactivity and power.

2.3 Smartphone Testbeds

There are currently several commercial platforms providing remote access to real smartphones, including *Sam-sung's Remote Test Lab* [33], *PerfectoMobile* [29], *De-vice Anyware* [19] and *AT&T ARO* [3]. These platforms differ from SmartLab in the following ways: i) they are mainly geared towards application testing scenarios on individual smartphones; and ii) they are *closed* and thus, neither provide any insights into how to efficiently build and run smartphone applications at scale nor support the wide range of functionality provided by SmartLab like sensors, mockups and automation.

Sandia National Laboratories has recently developed and launched *MegaDroid* [36], a 520-node PC cluster worth \$500K that deploys 300,000 AVD simulators. MegaDroid's main objective is to allow researchers to massively simulate real users. Megadroid only focuses on AVDs while SmartLab focuses on both ARDs and AVDs as well as the entire management ecosystem, providing means for *fine-grained* and *low-level* interactions with real devices of the testbed as opposed to virtual ones.

2.4 People-centric Testbeds

There is another large category of systems that focuses on opportunistic and participatory smartphone sensing testbeds with real custodians, e.g., *PRISM* [13], *Crowd-Lab* [12] and *PhoneLab* [4], but those are generally complementary as they have different desiderata than SmartLab.

Let us for instance focus on PhoneLab, which is a participatory smartphone sensing testbed that comprises of students and faculty at the University of Buffalo. PhoneLab does not allow application developers to obtain screen access, transfer files or debug applications, but only enables programmers to initiate data logging tasks in an offline manner. PhoneLab is targeted towards data collection scenarios as opposed to fine-grained and low-level access scenarios we support in this work, like deployment and debugging. Additionally, PhoneLab is more restrictive as submitted jobs need to undergo an Institutional Review Board process, since deployed programs are executed on the devices of real custodians.

Finally, UC Berkeley's Carat project [28] provides collaborative energy diagnosis and recommendations for improving the smartphone battery life from more than half a million crowd-powered devices. SmartLab is complementary to the above studies as we provide insights and micro-benchmarking results for a variety of modules that could be exploited by these systems.

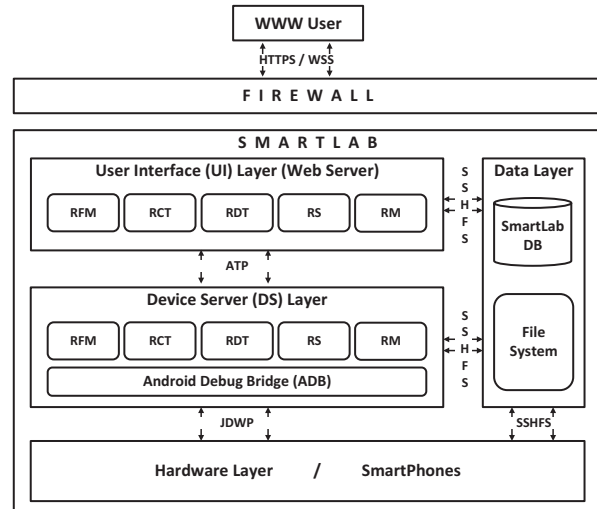


Figure 2: **The components of the SmartLab Architecture:** We have implemented an array of modules wrapped around standard software to bring forward a comprehensive smartphone testbed management platform.

3 SmartLab Architecture

In this section, we overview the architecture of our testbed starting out from the user interface and data layer moving on to the device server layer and concluding with the hardware layer, as illustrated in Figure 2. We conclude with an overview of our security measures and design principles.

3.1 User Interface and Data Layers

Interaction Modes: *SmartLab* implements several modes of user interaction with connected devices (see Figure 2, top-left layer) using either *Websocket-based interactions* for high-rate utilities or *AJAX-based interactions* for low-rate utilities. In particular, SmartLab supports: i) *Remote File Management (RFM)*, an AJAX-based terminal that allows users to push and pull files to the devices; ii) *Remote Control Terminals (RCT)*, a Websocket-based remote screen terminal that mimics touchscreen clicks and gestures but also enables users recording automation scripts for repetitive tasks; iii) *Remote Debug Tools (RDT)*, a Websocket-based debugging extension to the information available through the Android Debug Bridge (ADB); iv) *Remote Shells (RS)*, a Websocket-based shell enabling a wide variety of UNIX commands issued to the Android Linux kernels of allocated devices; v) *Remote Mockups (RM)*, a Websocket-based mockup subsystem for feeding ARDs and AVDs with GPS or sensor data traces encoded in XML for trace-driven experimentation.

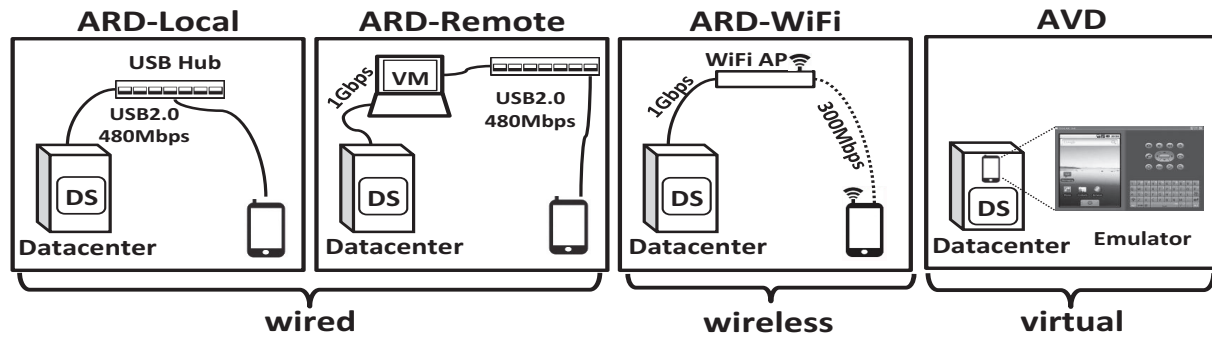


Figure 3: **Connection Modalities supported by SmartLab.** **ARD-Local:** Android Real Device (ARD) mounted locally to the Device Server (DS) through USB; **ARD-Remote:** ARD mounted through a USB port on a gateway PC to DS through a wired network; **ARD-WiFi:** ARD connected to DS through a WiFi AP; and **AVD:** Android Virtual Device running on DS.

WebSockets/HTML5: In order to establish fast and reliable communication between the *User Interface* and the underlying *Device Server (DS)*, SmartLab has adopted the HTML5 / WebSockets (RFC6455) standard thus enabling bi-directional and full-duplex communication over a single TCP socket from within the web browser. WebSockets are preferable for high-rate HTTP interactions, necessary in certain SmartLab subsystems, as opposed to AJAX calls that are translated into individual HTTP requests. WebSockets comprise of two parts: i) *an HTTP handshake*, during which certain application-level protocol keys are exchanged; and ii) *a data transfer phase*, during which data frames can be sent back and forth in full-duplex mode.

Currently, there are different types of WebSocket handshakes implemented by web browsers (e.g., Hixie-75, 76, 00 and HyBi-00, 07 and 10). In order to support websockets on as many browsers as possible, we have modified an open-source plugin, the kanaka websocketify plugin⁴ formerly known as wsproxy, part of the noVNC project. The given plugin takes care of the initial WebSocket handshake from within the browser but also shifts over to an SWF implementation (i.e., Adobe Flash), in cases where a browser is not HTML5-compliant, enabling truly-compliant cross-browser compatibility.

File System: SmartLab currently utilizes a standard ext4 local file system on the webserver. Upon user registration, we automatically and securely create a */user* directory on the webserver with a given quota. Our filesystem is mounted with *sshfs* to all DS images running in our testbed, enabling in that way a unified view of what belongs to a user. In respect to the connectivity between the filesystems of smartphones, we currently support two different options: i) mounting the */user*

directory on the devices with *sshfs*; and ii) copying data from/to the devices through *ADB*. The former option is good for performance reasons, but it is only available on Android 4.0 ICS, which provides in-kernel support for user-space filesystems (i.e., FUSE). On the contrary, the latter option is more universal, as it can operate off-the-shelf and this will be the major focus in this work.

SSHFS & MySQL: Communication between the web server file system and the device server's remote file system is transparently enabled through the SSHFS protocol. The same protocol can also be utilized for offering a networked file system to the smartphones, as this will be explained later in Section 6. Finally, the web server also hosts a conventional MySQL 5.5 database utilized for storing data related SmartLab users, devices and remote device servers.

3.2 Device Server (DS) Layer

Overview: DS is the complete Linux OS image having the SmartLab subsystems and ADB installed, which connects an ARD or AVD to our *User Interface (UI)*. Currently, we are using CentOS 6.3 x64 with 4x2.4GHz virtual CPUs, 8GB RAM, 80GB hard disk for our images. User interface requests made through Websockets reach DS at a Prethreaded Java TCP Server with *Non-blocking I/O* and *logging*.

ATP: We have implemented a lightweight protocol on top of websockets, coined ATP (*ADB Tunnel Protocol*) for ease of exposition, in order to communicate DS data to the UI and vice-versa (see Figure 2). *Downward ATP* requests (from UI to DS) are translated into respective calls using the *ddmilib.jar* library (including *AndroidDebugBridge*) for file transfers, screen capture, etc., as well as *monkeyrunners* and *chimpchat.jar*

⁴Kanaka, <https://github.com/kanaka/websocketify>

for disseminating events (e.g., UI clicks). Alternatively, one *Downward ATP* request might also yield a stream of *Upward ATP* responses, as this is the case in our screen capturing subsystem (i.e., one screenshot request yields a stream of images) presented in Section 6.2.

Device Plug-n-Play: Physically connecting and disconnecting smartphones from DS should update the respective UI status as well. Consequently, we've exploited the respective `AndroidDebugBridge` interface listeners, issuing the SQL statements to our MySQL database and updating the device status changes on our website.

DS Limitations: Currently, each DS can only support up to 16 AVDs and theoretically up to 127 ARDs, due to limitations in the ADB server that will be presented in Section 5. In order to support a larger number of connected devices with the current ADB release, we utilize multiple-DSs on each physical host of our datacenter each connecting 16 devices (ARDs or AVDs). This design choice is inspired from cloud environments and shared-nothing architectures deployed by big-data testbeds providing linear scalability by linearly engaging more resources.

DS Administration: In addition to the custom made Java Server each DS is also equipped with Apache and PHP. The local web server is responsible to host the administrative tools required for maintenance purposes similarly to routers and printers.

3.3 Hardware Layer

Hardware & OS Mix: SmartLab's hardware comprises both of *Android Smartphones* and our *Datacenter*. The latter encompasses over 16TB of RAID-5 / SSD storage on an IBM X3550 as well as 320GB of main memory on 5 IBM / HP multiprocessor rackables. We additionally deploy over 40 Android smartphones and tablets from a variety of vendors (i.e., HTC, Samsung, Google, Motorola and Nokia). The majority of our smartphones came with pre-installed Android 2.1-2.3 (Eclair, Froyo, Gingerbread). These devices were "rooted" (i.e., the process of obtaining root access) and upgraded to Android 4.0.4 (Ice Cream Sandwich), using a custom XDA-Developers ROM, when their warranty expired. Notice that warranty and rooting are claimed to be irrelevant in Europe⁵.

In SmartLab, rooted devices feature more functionality than non-rooted devices. Particularly, rooted devices in SmartLab can: i) mount remote filesystems over ssh; ii) provide a richer set of UNIX shell commands; and iii) support a higher performance to the screen capturing system by incorporating compression. Nevertheless,

SmartLab has been designed from ground up for non-rooted devices, thus even without applying the rooting process will support all features other than those enumerated above.

Physical Connections: We support a variety of connection modalities (see Figure 3) that are extensively evaluated in Sections 4 and 5. In particular, most of our devices are connected to the server in ARD-Local mode, utilizing USB hubs, as this is explained in Section 4. Similarly, more smartphones are also connected from within our research lab, in the same building, using the ARD-Remote mode.

This mode is particularly promising for scenarios we want to scale our testbed outside the Department (e.g., *ARD-Internet mode*, where latencies span beyond 100ms), which will be investigated in the future. Finally, a few devices within the Department are also connected in ARD-WiFi mode, but additional devices in this mode can be connected by users as well.

3.4 Security Measures

Security is obviously a very challenging task in an environment where high degrees of flexibility to users are aimed to be provided. In this section, we provide a concise summary of how security is provided in our current environment.

Network & Communication: SmartLab DS-servers and smartphones are located in a DMZ to thwart the spread of possible security breaches from the Internet to the intranet. Although nodes in our subnet can reach the public Internet with no outbound traffic filtering, inbound traffic to smartphones is blocked by our firewall. Interactions between the user and our Web/DS servers are carried out over standard HTTPS/WSS (Secure Websockets) channels. DS-to-Smartphone communication is carried out over USB (wired) or alternatively over secured WiFi (wireless), so we increase isolation between users and the risk of sniffing the network.

Authentication & Traceability: Each smartphone connects to the departmental WiFi using designated credentials and WPA2/Enterprise. These are recorded in our SQL database along with other logging data (e.g., IP, session) to allow our administrators tracing users acting beyond the agreed "Use Policy".

Compromise & Recovery: We apply a resetting procedure every time a user releases a device. The resetting procedure essentially installs a new SmartLab-configured ROM to clear settings, data and possible malware/ROMs installed by prior users. Additionally, our DS-resident home directory is regularly backed up to prevent accidental deletion of files. Finally, users have the choice to shred their SDCard-resident data.

⁵Free Software Foundation Europe, <http://goo.gl/fZZZQe>

3.5 Design Methodology/Principles

SmartLab’s architecture focuses on a number of desiderata such as *modularity*, *openness*, *scalability* and *expandability*. Its design was developed using a “greedy” bottom-up approach; in each layer/step, all alternative options that were available at the time it was designed were taken into consideration and the most efficient one was implemented. This was primarily because the research community long craved for the ability to test applications on real smartphone devices at the time SmartLab was designed. Because of this, we believed that there was no abundant time to dedicate for designing a *clean slate* architecture, like PlanetLab [30] and other similar testbeds. Additionally, some of the software/hardware technologies pre-existed in the laboratory and there was limited budget for upgrades. However, careful consideration was taken for each design choice to provide flexibility in accommodating the rapid evolution of smartphone hardware/software technologies.

4 Power and Connectivity

In this section, we present the bottom layer of the SmartLab architecture, which was overviewed in Section 3.3, dealing with power and connectivity issues of devices. In particular, we will analyze separately how *wireless* and *wired* devices are connected to our architecture using a microbenchmark that provides an insight into the expected performance of each connection modality.

4.1 Wired Devices

SmartLab *wired* devices (i.e., ARD-Local and ARD-Remote) are powered and connected through D-Link DUB-H7 7x port USB 2.0 hubs inter-connected in a *cascading* manner (i.e., “daisy chaining”), through standard 1.8mm USB 2.0 A-connectors rated at 1500mA. One significant advantage of daisy chaining is that it allows overcoming the limited number of physical USB ports on the host connecting the smartphones, reaching theoretically up-to 127 devices.

On the other hand, this limits data transfer rates (i.e., 1.5 Mbps, 480 Mbps and 5 Gbps for USB 1.0, 2.0 and 3.0, respectively). D-Link DUB-H7 USB 2.0 hubs were selected initially because they guarantee a supply of 500mA current on every port at a reasonable price, unlike most USB hubs available on the market. At the time of acquisition though, we were not sure about the exact incurred workloads and USB 3.0 hubs were not available on the market either.

USB 3.0: SmartLab is soon to be upgraded with USB 3.0 hubs that will support higher data transfer rates than USB 2.0. This is very important as in the experiments

of Section 6.2, we have discovered that applications requiring the transfer of large files are severely hampered by the bandwidth limitation of USB 2.0 hubs (max. 480 Mbps). We have already observed that newer hubs on the market are offering dedicated fast-charging ports (i.e., 2x ports at 1.2A per port and 5x standard ports at 500mA per port) in order to support more energy demanding devices such as tablets.

Power-Boosting: Instead of connecting 6x devices plus 1x allocated for the next hub in the chain, we have decided to use 3x Y-shaped USB cables in our release. This allows ARDs to consume energy from two USB ports simultaneously (i.e., 2x500mA), similarly to high-speed external disks, ensuring that the energy replenishment ratio of smartphones will not become negative (i.e., battery drain) when performing heavy load experiments such as stress testing or benchmarks (e.g., AnTuTu) on certain Tablets (e.g., Galaxy Tab drew up to 1.3A in our tests). A negative replenishment ratio might introduce an erratic behavior of the smartphone unit, failure to function, or overloading/damaging the ports.

Power Profiling: In order to measure physical power parameters in our experiments, we employed the Plogg smart meter plug connected to the USB hub, which transmits power measurements (i.e., Watts, kWh Generated, kWh Consumed, Frequency, RMS Voltage, RMS Current, Reactive Power, VARh Generated, VARh Consumed, and Phase Angle) over ZigBee to the *DS*. These measurements are provided on-demand to the *DS* administrator through the Administrative Tools subsystem. Additionally, we have installed a USB Voltage/Ampere meter (see Figure 1 top-left showing 4.67V), offering on-site runtime power measurements of running applications.

4.2 Wireless Devices

In our current setup, *wireless* devices (i.e., ARD-WiFi) are operated by the SmartLab research team that powers the devices when discharged. Additionally, users can connect their own device remotely and these will be privately available to them only (e.g., see Figure 14 center). This particular feature is expected to allow us offering a truly programmable wireless fleet in the near future, as this is explained in Section 8. In this subsection, we will overview the underlying logistics involved in getting a wireless device connected to SmartLab over wireless ADB. Note that this process is automated through the SmartLab UI. In particular, the following commands have to be issued on rooted devices such that a smartphone can accept commands from the device server:

```
# On Smartphone (rooted):  
# Enable ADB over wireless  
#(to disable set port -1):
```



```

setprop service.adb.tcp.port 5555
stop adbd
start adbd
# On PC:
adb connect <device-ip>:5555

```

4.3 Connectivity Microbenchmark

In order to evaluate the time-efficiency of various connection modalities (i.e., wired or wireless) to our *DS*, we have performed a microbenchmark using wired ARDs (i.e., ARD-Local and ARD-Remote) and wireless ARDs (i.e., ARD-WiFi). The wireless connectivity is handled by a 802.11b/g/n wireless router (max. 300 Mbps) deployed in the same room as the ARDs and connected directly to the *DS*.

Those experiments were conducted for calculating the time needed for transferring 2.5MBs to up to 16 devices. As we already mentioned in Section 3.2, those results can be generalized to larger configurations by increasing the number of *DS* images. In our experimentation, we observed that ARD-WiFi features the worst time compared to the other two alternatives. For example, in the case of 16 ARDs, the time required for sending the file reaches 12 seconds as opposed to 4.8 seconds and 1.4 seconds for ARD-Remote and ARD-Local, respectively, as this is summarized in Table 1. One reason for this is because the cascading USB 2.0 hubs offer much higher transfer rate (max. 480Mbps) than the wireless router, which never reached over 130Mbps.

Table 1: Transferring a 2.5MB file to 16 Devices

Connectivity Mode	Average Time (10 trials)
ARD-Local	1.4 seconds
ARD-Remote	4.8 seconds
ARD-WiFi	12 seconds

Another observation is that ARD-Local devices outperform ARD-Remote devices, as the former are locally mounted to *DS*, thus avoid the overhead of transferring data via a network. Yet, ARD-Remote devices are particularly promising for scaling our testbed outside the server room, thus are considered in this study.

5 Android Debug Bridge (ADB)

In this section, we provide an in-depth understanding of the *Android Debug Bridge* (ADB), which handles the bulk of communication between the connected smartphones and the *Device Server* (*DS*) (see Figure 2).

The ADB command (version 1.0.31, in this study) is part of the platform tools (version 16.0.1, in this study), provided by the Android development tools enabling the

Android Development Tools		
/tools (as of 21.0.1)		
SDK Tools	Debuggers/Emulators	android, ddms, emulator
	Testers/Stress Testers	monkey, monkey runner
	Profilers	dmtracedump, systrace, traceview, hprof-conv
	Graphical/Layout Optimizers	hierarchyviewer, draw9patch, layout-opt
	.apk tools	proguard, zipalign
	Miscellaneous	mksdcard, sqllite3
/platform-tools (as of 16.0.1)		
Platform Tools	Debuggers	adb
	Shell tools	shell, bmgr, logcat
	Miscellaneous	aidl, aapt, dexdump, dx

Figure 4: The Android Development Tools.

development, deployment and testing of applications using ARDs and AVDs. These tools are classified into two categories (see Figure 4): i) *the SDK tools*, which are platform-independent; and ii) *the Platform tools*, which are customized to support the features of the latest Android platform.

In the latter category, there are also some shell tools that can be accessed through ADB, such as *bmgr*, which enables interaction with the backup manager of an Android device, and *logcat*, which provides a mechanism for collecting and viewing system debug output. Additionally, there platform tools such as *aidl*, *aapt*, *dexdump*, and *dx* that are typically called by the Android build tools or Android development tools.

5.1 Debugging Android Applications

Android applications can be developed using any Android-compatible IDE (e.g., Eclipse, IntelliJIDEA, Android Studio) and their code is written using the JAVA-based Android SDK. These are then converted from Java Virtual Machine-compatible (.class) files (i.e., bytecode) to *Dalvik-compatible Executables* (.dex) files using the *dx* platform tool, shrunk and obfuscated using the *proguard* tool and ported to the device using the *adb install* command of ADB. The compact .dex format is specifically tailored for systems that are constrained in terms of memory and processor speed.

As illustrated in Figure 5 (right), each running application is encapsulated in its own process and executed in its own virtual machine (DalvikVM). Additionally, each DalvikVM exposes a single unique port ranging from 8600-8699 to debugging processes running on both local and remote development workstations through the *ADB*

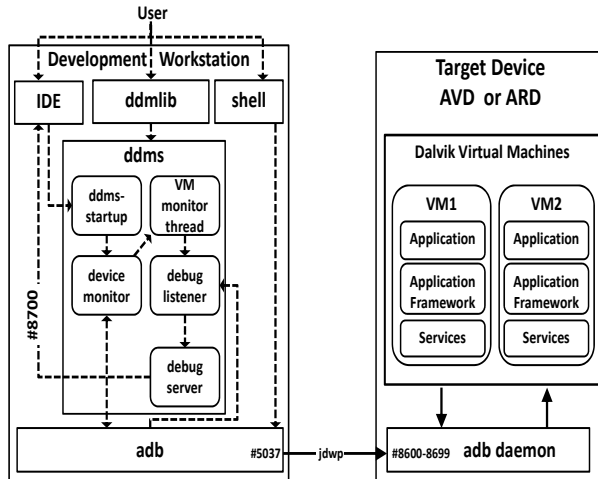


Figure 5: **Android Debug Bridge (ADB)**. Overview of components involved in the debugging/deployment process of Android applications.

daemon (*adb*) shown in Figure 5 (left). The *adb* is running on each target device and facilitates the connection with server processes (i.e., stream sockets) through the *java debug wire protocol* (*jdwp*-*adb*) transport protocol.

Debugging applications can be accomplished through the *Dalvik Debug Monitor Server* (DDMS), or its superset tool *Android Device Monitor*. DDMS can be executed as: i) a stand-alone application (*ddms.jar*); ii) initiated by the IDE; or iii) embedded to a java application (*ddmlib.jar*). All communication between DDMS and target devices is handled via ADB, which deploys a background process on the development machine, i.e., the *ADB server* that manages communication between the ADB client and the ADB daemon.

5.2 ADB Pipelining Microbenchmark

As mentioned in the previous section, users must connect directly on ADB or through a mediator library such as *ddmlib*, *monkeyrunner* or *chimpchat* both of which connect to ADB, in order to perform any action on a target device. However, initiating individual ADB connections for each action introduces a significant time overhead as it involves scanning for existing connections or creating new connections each time.

In order to justify this, we have conducted a microbenchmark using the Android *chimpchat* SDK tool, which allows amongst other functionality propagating events (e.g., mouse clicks) to a target device. More specifically, we generate 100 mouse click events and distribute them up to 16 ARD-Locals using two different settings: i) a new connection is initiated for each ADB call, denoted as *No Pipelining* (np); and ii) a single

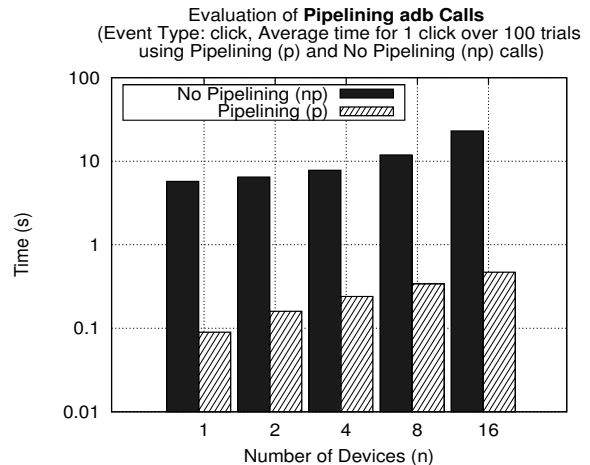


Figure 6: **ADB Pipelining Microbenchmark**. Evaluating the average time for one click with *pipelining* and *no-pipelining*. SmartLab utilizes *pipelining*.

persistent connection is utilized for pipelining all ADB calls, denoted as *Pipelining* (p). The latter can be accomplished through the creation of a connection at the start of the script and then utilizing that connection for all propagated events. Note that the reason we have selected mouse click events is because they are extremely lightweight and do not introduce other time-demanding overheads (e.g., I/O), thus allowing us to focus on the time-overhead incurred by each connection when pipelining ADB calls or not.

Figure 6 shows the results of our evaluation (averaged over 100 trials). We observe that the overhead of not pipelining ADB calls is extremely high. In particular, 1 click on 16 AVDs using no-pipelining requires 23s, as opposed to pipelining that only requires 0.47s (i.e., a 98% improvement.) Such extreme time overheads may be a prohibiting factor for some applications, thus careful consideration must be taken to ensure that applications communicate with target devices through a single connection.

In SmartLab, we utilize a single persistent ADB connection for each particular component (e.g., separate ADB for Screen Capture and Shell Commands.) Through the persistent connection, all ADB calls are pipelined thus alleviating the aforementioned inefficiency. The above configuration offloads the issue of concurrent ADB calls to a single device from different components, to ADB and the OS, as a device is allocated to only one user at-a-time (thus high concurrency patterns are not an issue.)

6 Device Server (DS)

In this section, we present the middle layer of the SmartLab architecture, which was overviewed in Section 3.2 and illustrated in Figure 2, dealing with device management. In particular, we will explain and evaluate the following subsystems: *Filesystem and File Management*, *Screen Capture*, *Logging*, *Shell Commands* and *Sensor/GPS Mockups*.

6.1 File Management (RFM) Subsystem

We start out with a description of the File Management UI and finally present some performance microbenchmarks for pushing a file and installing an application on a device using ADB pipelining.

Remote File Management (RFM) UI: We have constructed an intuitive HTML5/AJAX-based web interface, which enables the management of the local filesystems on smartphones *individually* but also *concurrently* (see Figure 7). In particular, our interface allows users to perform all common file management operations in a streamlined manner. The RFM interface starts by launching a separate window for each AVD or ARD that is selected by the user and displays a tree-based representation of its files and directories under the device's `/sdcard` directory. Similarly, it launches two additional frames (i.e., JQuery dialogs): i) one frame displays the users' "Home" directory (top-left); and ii) another frame displays a `/share` directory, which is illustrated in Figure 7 (top-center). The user is then able to move a single file or multiple files to multiple target devices.

The File Management subsystem is also responsible for replicating any files moved to the `/share` directory to each target device's `/sdcard/share` directory. Furthermore, an *Update All* button and a *Push All* button have been placed below the `/share` directory in order to support simultaneous updating or merging the `/share` directory on existing and newly reserved devices. In order to accomplish these operations, the RFM UI issues separate web requests, which include: i) the target device id (or multiple devices ids); ii) the absolute location of a single file (or multiple files); and iii) the type of operation. Requests are transmitted using AJAX, to the device server, which is responsible to execute the appropriate `adb push` and `adb pull` commands to transfer files to or from a device, respectively, all over the *ATP* protocol discussed earlier.

File-Push Microbenchmark: The time required to transfer files from and to target devices differs significantly according to the type of device. In order to investigate this, we have conducted a microbenchmark that measures the time overhead for transferring files to/from the aforementioned different types of target de-

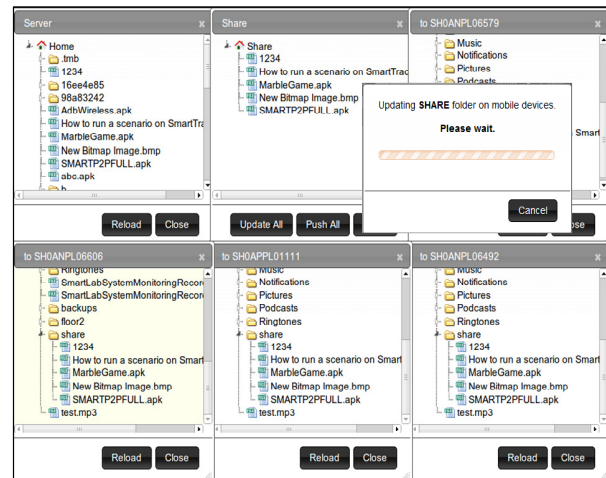


Figure 7: **Remote File Management (RFM) UI.** A share folder enables to push/pull files to devices concurrently. FUSE-enabled devices can feature sshfs shares.

vices. More specifically, we have utilized a 10MB file and distributed this file to up to 16 AVDs, ARD-WiFi, ARD-Remote and ARD-Local, respectively. The ARD-WiFi devices were assigned to students that were moving around our department premises in order to provide a realistic mobility scenario. Each experiment was executed 10 times and we recorded the average at each attempt.

The results are shown on the left side of Figure 8, which clearly illustrates the advantage of using ARD-Local devices in experiments requiring large amounts of data to be transferred to devices (e.g., large trajectory datasets). Additionally, the results show that the disk I/O overhead introduced by the usage of the emulated devices (i.e., AVDs) justifies the linearly increasing amount of time for transferring files on those devices. In the case of remotely connected ARDs (ARD-Remote) the large time delays are attributed to communicating over the network. Finally, the ARD-WiFi devices feature the worst time overhead because the file transfer is hampered by the wireless network's low bandwidth in mobility scenarios.

File-Install Microbenchmark: In order to examine the cost of installing applications, which include transferring the application file (.apk) and its installation, we have conducted another microbenchmark that calculates the required time. Similarly to the previous experimental setting, we measure the time for transferring and installing a sample application of typical 1MB size, to each type of target devices. The results are shown on the right side of Figure 8. We observe that transferring and installing the selected sample application introduces an additional time overhead. For example, in the 1x target device scenario, the sample application requires a total of $\approx 2.2s$

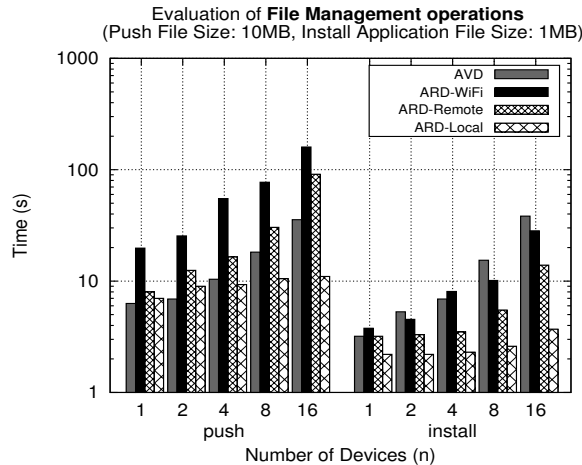


Figure 8: **File Management Microbenchmark.** Evaluating the average time for transferring files and installing applications on different types of target devices.

from which 0.7s accounts for file transfer and 1.5s for installing the application. The results provide a clear indication that emulated devices are not the appropriate type of Android devices for performing I/O intensive experiments such as evaluating network performance and database benchmarking. Additionally, the sample application utilized in the experiments did not perform any specialized deployment functions during setup (e.g., extracting other files, installing database), thus its installation overhead is minimal. The time required for installing more complex applications varies greatly according to the requirements of the application.

6.2 Screen Capture (RCT) Subsystem

The Screen Capture subsystem enables capturing the actual screen of a target device so that it can be displayed to the user through the *Remote Control Terminal (RCT)* UI component (see Figure 9). Additionally, it supports a variety of events using the *chimpchat library* such as: i) *control events* (e.g., power button, home button); ii) *mouse events* (e.g., click, drag); and iii) *keyboard events* (e.g., key press).

Screen-Capture Alternatives: Capturing a screenshot of an ARD or AVD can be accomplished through the following means (either directly or through an application): i) on ARDs using the `cat` command (`/dev/fb0` or `dev/graphics/fb0` according to the target device version) and redirecting the output to an image file; ii) on both using the Android `monkeyrunner` script command `takeSnapshot()`; iii) on both by continuously invoking the `getScreenShot()` command provided by the `ddmlib` library; and iv) on both similarly

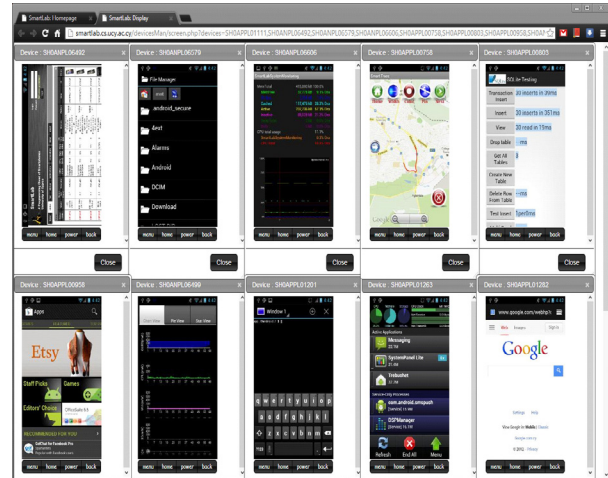


Figure 9: **Remote Control Terminal (RCT) UI.** Our implementation allows concurrent interaction (e.g., clicks, drag gestures, key press) on multiple devices.

to (iii), by continuously listening to the direct stream that contains the contents of each consecutive screenshot (i.e., `readAdbChannel()` in `ddmlib`). The SmartLab screen capture component has been developed using the (iv) approach because it is more efficient both in terms of memory and time as it utilizes buffered-oriented, non-blocking I/O that is more suitable for accessing and transferring large data files as shown next.

Screen-Capture Microbenchmarks: In order to justify our selection, we have performed a microbenchmark that evaluates the time required to generate and transfer 100 consecutive screenshots from up to 16 ARD-Local devices using the (ii) and (iv) approaches denoted as *monkeyrunner python scripts* and *Screen Capture*, respectively. Approaches (i) and (iii) were omitted from the experiment because the former cannot provide a continuous stream of screenshots required by RCT and the latter does not provide any guarantee that a screenshot image will be ready when the `ddmlib` library's `getScreenShot()` command is invoked, which may lead to presentation inconsistencies. The experiment was performed only on ARD-Local devices that outperform AVD, ARD-Remote and ARD-WiFi devices w.r.t. file transfer operations as is the case of capturing and displaying a screenshot image.

The results of our microbenchmark, depicted in Figure 10 (left), clearly justify our selection. In particular, SmartLab's Screen Capture subsystem always maintains a competitive advantage over *monkeyrunner python scripts* for all number of target devices. Additionally, we notice that the time required for processing images for up to 8 devices is almost identical at 0.97 ± 0.03 s. However, when 16 devices are used, the time required

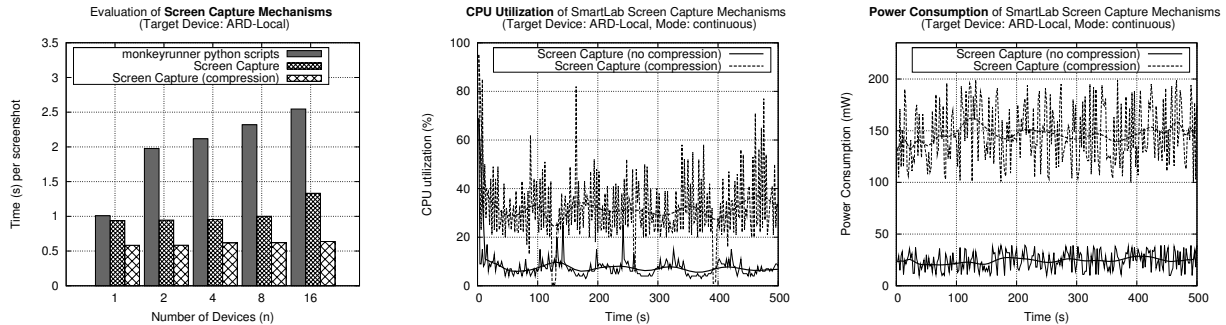


Figure 10: **Screen Capture Subsystem Microbenchmarks** on ARD-Local devices. **(left)** Evaluating the time overhead of capturing screenshot images using monkeyrunner python scripts and SmartLab’s Screen Capture subsystem; Evaluation of SmartLab Screen Capture compression mechanism w.r.t.: **(center)** CPU Utilization.; and **(right)** Power Consumption.

for processing the screenshot images increases by $\approx 30\%$ (i.e., $1.33s \pm 0.6s$). This may be inefficient in the case of applications requiring real-time control of the target devices. On the other hand, for automated bulk tests the above is not a big problem, as these are not affected by the interaction latency that is inherent in any type of remote terminal like ours. By performing a number of file transfer benchmarks on our USB 2.0 hubs, we discovered that this happens because the maximum available bandwidth for file transfer was approximately 250Mbps (theoretically up-to 480Mbps). Consequently, it was not adequate to support the necessary 320Mbps USB bandwidth incurred by the 16 devices each transferring a 480×800 screenshot with an approximate size of 2.5MB per shot (i.e., $16 \times 2.5MB \times 8bps = 320Mbps$).

On-Device Compression: Currently, producing large screenshot files cannot be avoided as there are no mechanisms for reducing the file size (i.e., compression). In order to alleviate this problem, we experimented with an in-house module for rooted devices that provides the ability to generate compressed screenshot images (e.g., JPEG, PNG) locally at the device prior to transmitting them over the network. We evaluated the revised Screen Capture subsystem, denoted *Screen Capture (compression)* using the same configuration as in the previous experiment.

We observe in Figure 10 (left) that the Screen Capture (compression) clearly outperforms the *Screen Capture (with no compression)*, as expected. This is because the files generated by Screen Capture (compression) never reached over 45KBs. As a result, the revised Screen Capture subsystem is not affected by the limitation of the USB 2.0 hub as the combined bandwidth rate required was 5.7Mbps (i.e., $16 \times 45KB \times 8bps$) and this is the reason why the time required per screenshot for all number of devices remains persistent at $0.6 \pm 0.05s$.

Power and CPU issues: Compressing images though, requires additional CPU effort as well as increased power consumption on a smartphone. In order to investigate these parameters, we have utilized a custom SmartLab System Monitor application (see Figure 9, third screenshot on top row for overview) and PowerTutor tools (see on the same figure the second screenshot on bottom row), in order to measure CPU utilization and power consumption, respectively. Our findings are illustrated in Figure 10 (center and right). We observe that the CPU utilization in the compression scenario reaches $28 \pm 15\%$ as opposed to $7 \pm 3\%$ when no compression is performed. This is important as applications requiring high CPU utilization should use the conventional (i.e., no compression) approach. Similarly, the power consumption of compression is higher. However, the difference is very low compared to other smartphone functions (e.g., 3G busy $\approx 900mW$ [8]). In the future, we aim to investigate automated techniques to switch between available screen capture modes.

6.3 Logging (RDT) Subsystem

The SmartLab Logging subsystem is responsible for parsing the debug data generated locally at each target device and providing comprehensive reports regarding the status of each target device to the user. The log data is generated automatically by the Android OS and includes various logs such as system data, system state and error logs. These can be accessed directly through the ADB commands `dumpsys`, `dumpstate`, and `logcat` respectively or through the `bugreport` command, which combines all previous logs into a comprehensive log file.

The Logging subsystem is accessible through the *Remote Debug Tools (RDT)* component of the web server. The logging process starts with the RDT component, which upon a user request for logs initiates a web request

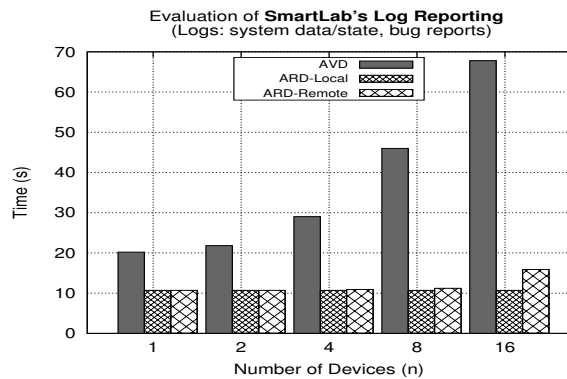


Figure 11: **Logging (RDT) Microbenchmark.** Time required for retrieving the report log from different target devices.

including the target device id (or multiple devices ids) using AJAX. The Logging subsystem receives this request and propagates an `adb bugreport` command to the target devices selected by the user. Consequently, the resulting log report is saved to a separate directory inside the user home directory and the user is able to choose whether to manually traverse the report or to use a more sophisticated tool such as the `ChkBugReport` tool⁶ that is able to illustrate the results in a comprehensive graphical manner. If the latter is chosen, the Logging subsystem invokes the `ChkBugReport` tool, passing the log report file as a parameter. Finally, the resulting HTML files generated by the `ChkBugReport` tool are stored in the users' "Home" directory.

Currently, the Logging subsystem (using `ChkBugReport`) extracts the following information: *i) Stacktraces; ii) Logs; iii) Packages; iv) Processes; v) Battery statistics; vi) CPU Frequency statistics; vii) Raw data; and viii) Other data.* Additionally, each `ChkBugReport` plugin can detect (possible) errors, which are highlighted in the errors section of the HTML report files. For instance, by looking at the Stack-trace section the user might observe deadlocks or strict mode violations in addition to other useful information.

We have conducted a microbenchmark in order to evaluate the time overhead for gathering log reports from the target devices. More specifically, we gathered the bugreports from up to 16 AVDs, ARD-Remote and ARD-Local devices, respectively. The results, shown in Figure 11 clearly illustrate that ARD-Remote and ARD-Local devices outperform AVDs. This confirms again that utilizing real devices can speed up the experimental process and produce output results more efficiently.

⁶Check Bug Report, <http://goo.gl/IRPUW>.

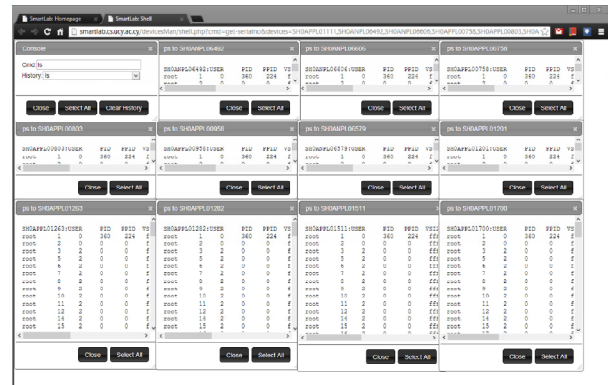


Figure 12: **Remote Shell (RS) UI.** Allows concurrent UNIX command executions on multiple devices.

6.4 Shell Commands (RS) Subsystem

The Shell Command subsystem works in collaboration with the web server's *Remote Shells (RS)* component (see Figure 12) in order to execute shell commands from SmartLab to all target devices selected by the user. These commands include every available `adb shell` command supported by the Android OS on rooted devices and a considerable subset on non-rooted devices. The RS component propagates each shell command through a bi-directional WebSocket to the Shell Commands subsystem, which in turn executes each command on the target devices and returns the resulting data back to the RS web interface. More specifically, it allows a user to select a set of target devices and launch a separate window consisting of frames (jQuery dialogs) for each target device. Each frame represents an interactive shell on the selected device allowing the user to remotely issue shell commands to single or multiple devices at the same time.

6.5 Sensor/GPS Mockup (RM) Subsystem

A mockup provides part of a system's functionality enabling testing of a design. In the context of Android, Mockup refers to the process of extending an AVD's or ARD's particular sensor or GPS with custom values. Additionally, one important benefit of Mockups is that these can support the addition of sensors that may not exist in the hardware of a particular ARD (e.g., NFC). The Android SDK 4.0 supports the mockup of GPS data through the following command sequence:

```
# On PC running AVD (5554: emulator)
telnet localhost 5554
geo fix latitude longitude
```

In order to support both GPS and other sensor mockups in SmartLab, (e.g., *accelerometer, compass, orienta-*



Figure 13: **Sensor/GPS Mockup (RM):** (left, center) A data trace of various sensor measurements encoded in XML. The given file can be loaded to ARDs and AVDs through this subsystem; (right) An application built with SLSensorManager using the measurements.

tion, temperature, light, proximity, pressure, gravity, linear acceleration, rotation vector and gyroscope sensors) on both ARDs and AVDs, we opted for a custom module.

In particular, we have been inspired by the *SensorSimulator*⁷ open source project, which establishes a socket server on *DS* feeding devices with sensor or GPS readings encoded in XML (see Figure 13 left). As this functionality is completely outside the ADB interaction stream, we were required to provide each application with a custom library, coined `SLSensorManager.jar`.

SLSensorManager Library: Our library can be embedded to any Android application enabling interaction with the SmartLab GPS/Sensor subsystem running on *DS*. For example, Figure 13 (right) shows how a sample application has been constructed with this library. In fact, our library has precisely the same interface with the Android SDK `SensorManager`, consequently a user can override Android’s default behavior very easily achieving in that way to feed its allocated device from a real and realistic sensor dataset.

Hardware Emulation: With Android Tools `r18` and Android 4.0, developers have the opportunity to redirect real sensor measurements, produced by the ARDs, to the AVDs for further processing. It is important to mention that this functionality is the reverse of what we are offering. In our case, we want to be able to redirect data from a text file to an ARD, such that a given experiment on ARDs or AVDs uses a data file to drive its sensors. Recording sensor readings to text files can be carried out very easily with a variety of tools.

⁷Openintents, <http://goo.gl/WkuN>

7 Experiences using SmartLab

In this section, we present four different research efforts, including: GPS-trajectory benchmarking [38], peer-to-peer search [22], indoor positioning [24] and database benchmarking (the last carried out in the context of a graduate course.) None of the following studies would have been feasible with AVDs, as all of the below scenarios require *fine-grained* and *low-level* access (e.g., sdcard, WiFi, real CPU and mobility).

7.1 Trajectory Benchmarking

SmartLab has been utilized in the context of the SmartTrace[38] project⁸, which is a prototype crowd-sourced trajectory similarity search framework enabling analytic queries over outdoor GPS traces and indoor WiFi traces, stored on users’ smartphones.

SmartLab was utilized to carry out a GPS mockup study with the GeoLife GPS Trajectories [39]. The SmartLab file management functionality was extremely useful in disseminating input traces and collecting our experimental results from the local sdcards of smartphones. Additionally, the Remote Control Terminals were equally important in order to setup and run the experiments. Finally the device diversity allowed us to test trajectory comparison algorithms on many smartphones (see Figure 14, left).

As SmartLab is currently firewalled (i.e., the device server is only accessible through the webserver), it is not feasible to have some outside process connect to the SmartLab smartphone processes internally. In order to overcome this correct security configuration, we wrote our Smarttrace smartphone clients in a manner that these only issued outgoing TCP traffic (i.e., connecting to the outside server) as opposed to incoming TCP traffic.

Finally, in order to scale our experiments to 200 smartphone processes, in the absence of such a large number, we launched 10 concurrent threads to each of our 20 reserved ARD devices.

7.2 Peer-to-Peer Benchmarking

SmartLab was also utilized in the context of a Peer-to-Peer benchmarking study (i.e., the SmartP2P [22] project). SmartP2P offers high-performance search and data sharing over a crowd of mobile users participating in a social network. Similarly to SmartTrace, the experimental evaluation of SmartP2P was performed on real devices reserved through SmartLab. A subtle difference of this study was that the UI interactions were recorded from within RCT into automation scripts stored on SmartLab. Those scripts, running on our Device

⁸SmartTrace, <http://smarttrace.cs.ucy.ac.cy/>



Figure 14: **Research with SmartLab.** (Left) Testing trajectory comparison algorithms on a diverse set of smartphones in SmartTrace [38]; (Center) Testing indoor localization using ARD-WiFi mode in Airplace [24]; (Right) Testing various SQLite tuning parameters in the context of an advanced databases course.

Server, would automatically repeat an experimental simulation improving automation and repeatability of the experimental evaluation process.

7.3 Indoor Localization Testing

WiFi-based positioning systems have recently received considerable attention, both because GPS is unavailable in indoor spaces and consumes considerable energy. In [24], we have demonstrated an innovative indoor positioning platform, coined Airplace, in order to carry out fine-grained localization with WiFi-based RadioMaps (i.e., 2-4 meters accuracy). SmartLab has facilitated the development, testing and demonstration of Airplace and its successor project Anyplace⁹ considerably as explained next.

Firstly, we extensively used the ARD-WiFi mode, which allowed us to move around in a building localizing ourselves while exposing the smartphone screen on a remote web browser through SmartLab (e.g., see Figure 14, center). The particular setting has proved considerably useful for demonstrations at conferences as the bulk of existing AndroidScreenCapture software are both USB-based, which hinders mobility, but are also inefficient as they provide no compression or other optimizations.

Secondly, SmartLab allowed us to collect and compare *Received Signal Strength (RSS)* indicators from different WiFi chip-sets, which is important for RSS measurements and would not be possible with AVDs. Finally, SmartLab allowed us to test the generated APK on a variety of devices.

7.4 DB Benchmarking

A recent study by NEC Labs America [20], has shown that underlying flash storage on smartphones might be

⁹Anyplace, <http://anyplace.cs.ucy.ac.cy/>

a bottleneck in many smartphone applications, which cache results locally.

In the context of an Advanced DB course at our department, students were asked to carry out an extensive experimental evaluation of SQLite, the most widely deployed SQL database engine in the world that is readily available by the Android SDK. One particular objective of this study was to find out how the reads and writes could be optimized. For the given task students parsed the sqlite data files stored by various smartphone apps in their sqlite dbs. Subsequently, students carried out a number of trace-driven experimentations.

Figure 14 (right) for example, shows how sequential inserts are affected by disabling the PRAGMA synchronous and PRAGMA journalmode runtime options on a smartphone storing its data on a FAT32-formatted sdcard. In respect to SmartLab, it is important to mention that APK and data trace files were seamlessly transferred to target devices. Additionally, after installing the APKs it was very efficient working on several RCT control terminals concurrently, carrying out the experimental study quickly.

8 Future Developments

In this section, we outline some of our current and future development plans:

8.1 Experimental Repeatability

Allowing seamless experimental repeatability and standardization is a challenging task for smartphone-oriented research. Looking at other research areas, somebody will realize that open benchmarking datasets and associated ground truth datasets have played an important role and academic and industrial research over the last decades. For instance, the TREC Conference series co-sponsored by National Institute of Standards and Tech-

nology (NIST) of the U.S. Commerce Department is heavily embarked by the information retrieval community. Similarly, the TPC (Transaction Processing Performance Council) non-profit corporation, founded to define transaction processing and database benchmarks, is heavily embarked by the data management community.

In the context of our project we are: i) collecting our own data on campus (e.g., WiFi RSS data [24]) and additionally trying to convince other research groups contributing their own data to the SmartLab repository. In respect to storage, we are using a prototype Apache HBase installation within our datacenter, to store sensor readings in a tabular and scalable (i.e., column-oriented) format.

Apache HBase is an open-source version of Google's Bigtable [7] work utilized to store and process Crawling data, Maps data, etc., without the typical ACID guarantees that are slowing and scaling down distributed relational databases (e.g., MySQL-Cluster-like DBs). The given store can be utilized to store billions of sensor readings that can be advantageous to our *GPS/Sensor Mockup* subsystem. This will allow a researcher to test an algorithm or application using tens or hundreds of smartphone devices using automated scripts, similarly to [36] but with bigger data. Another envisioned scenario would be to enable smartphone experimentation repeatability and standardization.

8.2 Urban-scale Deployment

We are currently working with local telecommunication authorities in order to obtain mobile data time for our mobile fleet and local transportation companies in order to have them move our devices around in a city, with possible free WiFi access to their customers as an incentive.

The envisioned scenario here is to be able to test an algorithm, protocol or application with ARD-Mobile devices in an urban environment, providing in that way an open mobile programming cloud. This could, for example, support data collection scenarios, e.g., *VTrack* [35], *CitySense* [27], and others, which rely on proprietary software/hardware configurations, but also online traffic prediction scenarios, trajectory and sensor analytics, crowdsourcing scenarios, etc.

Such ARD-Mobile devices need of course limiting the capabilities of users (e.g., prohibit the installation of custom ROMs, disable camera, sound and microphone.) We are addressing this with a customized after-market firmware distribution for Android (i.e., ROM), named CyanogenMod¹⁰. We did not opt for the *Android Open Source Project* (AOSP), as it was fundamentally difficult to port the drivers of all ARD we have ourselves.

¹⁰CyanogenMod, <http://www.cyanogenmod.org/>

Moreover, notice that the AOSP project currently supports only the Google Nexus family¹¹ of phones off-the-shelf. Enabling urban sensing scenarios also has a legal dimension as Europe has a strict Data Protection Policy (e.g., Directive 95/46/EC on the protection of individuals with regard to the processing of personal data and on the free movement of such data.)

8.3 Web 2.0 API

We are currently working on a Web 2.0 JSON-based API of our testbed using the Django framework¹². Django comes with rich features including a *Model-View-Controller* (MVC) architecture that separates the representation of information from the users' interaction with it. In particular, this effort will allow users to access the subsystems of our testbed in a programmable manner (i.e., Web 2.0 JSON interactions) and write applications to extend SmartLab, similarly to NagMQ [32].

Consider for instance the Eclipse IDE, which we are currently extending with Smartlab integration functionality through its API. The high level idea here is to allow developers to compile their code and deploy it immediately on available devices accessible on SmartLab, by having the Smartlab UI become part of the Eclipse IDE.

Finally, we are considering the integration with Google App Inventor¹³, such that programmers can see their developments immediately on SmartLab.

8.4 Federation Issues and PG Management

Our Web 2.0 API will allow us to implement SmartLab federation scenarios. For example, groups around the globe can interface with SmartLab enabling a truly global smartphone programming cloud infrastructure. Additionally, we aim to develop a SmartLab derivative for *Personal Gadget* (PG) management, which was motivated in the introduction. This will be facilitated by the fact that personal gadgets are quantitatively and qualitatively increasing but more importantly, by the fact that PGs are *reusable* after they become deprecated as they are *programmable* and *feature-rich*.

8.5 Security Studies

SmartLab can be utilized in order to conduct experiments related to enhanced smartphone security and privacy. SmartLab smartphones can be used as honey pots for investigating intruders' behavior. Additionally, smartphones can be used as replicas of real devices enabling

¹¹Nexus Factory Images, <http://goo.gl/v1Jwd>

¹²Django Framework, <https://www.djangoproject.com/>

¹³MIT AppInventor, <http://appinventor.mit.edu/>

the replication of real event execution performed on real devices. As a result, researchers can use SmartLab in order to identify newly introduced threats by gathering statistics from multiple replicas. Furthermore, SmartLab can be utilized in the context of projects using replicated execution [31] for validation and verification purposes. Carefully investigating the security aspects related to SmartLab will be a topic of future research. At the end, SmartLab's administrators will be able to present their experiences related to managing security in a distributed mobile environment similarly to the work presented by Intel on how to secure PlanetLab [6].

9 Conclusions

In this paper, we have presented the first comprehensive architecture for managing a cluster of both real and virtual Android smartphones. We cover in detail the subsystems of our architecture and present micro-benchmarks for most of the internally components.

Our findings have helped us enormously in improving the performance and robustness of our testbed. In particular, by pipelining Android Debug Bridge (ADB) calls we managed to improve performance by 98%. Additionally, by compressing screen capture images with moderate CPU overhead we improve capturing performance and minimize the network overhead.

This paper has also presented four different research and teaching efforts using SmartLab, including: GPS-trajectory benchmarking, peer-to-peer search, indoor positioning and database benchmarking. Finally, this paper has overviewed our ongoing and future SmartLab developments ranging from Web 2.0 extensions to urban-scale deployment primitives and security.

Our long-term goal is to extend our testbed by engaging the research community that can envision and realize systems-oriented research on large-scale smartphone allocations but also enable a platform for *Personal Gadget (PG)* management.

Acknowledgments

We would like to thank Matt Welsh (Google) and Stavros Harizopoulos (HP Labs) for the useful discussions that lead to the realization of this work. Finally, we would like to thank our USENIX LISA'13 shepherds, Carolyn Rowland and Adele Shakal, and the anonymous reviewers for their insightful comments. This work was financially supported by the last author's startup grant, funded by the University of Cyprus. It has also been supported by EU's COST Action IC903 (MOVE), by EU's FP7 MODAP project and EU's FP7 Planetdata NoE.

References

- [1] Tarek Abdelzaher, Yaw Anokwa, Peter Boda, Jeff Burke, Deborah Estrin, Leonidas Guibas, Aman Kansal, Sam Madden, and Jim Reich. "*Mobiscopes for Human Spaces*", IEEE Pervasive Computing, Volume 6, Issue 2, April 2007.
- [2] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. "*FAWN: A fast array of wimpy nodes*", In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (*SOSP'09*). ACM, New York, NY, USA, 1-14, 2009.
- [3] AT&T Application Resource Optimizer (ARO), Free Diagnostic Tool: <http://goo.gl/FZnXS>
- [4] Rishi Baldawa, Micheal Benedict, M. Fatih Bulut, Geoffrey Challen, Murat Demirbas, Jay Inamdara, Taeyeon Ki, Steven Y. Ko, Tevfik Kosar, Lokesh Mandvekar, Anandathirya Sathiyaraja, Chunming Qiao, and Sean Zawicki. "*PhoneLab: A large-scale participatory smartphone testbed (poster and demo)*", 9th USENIX conference on Networked systems design & implementation (*NSDI'12*). USENIX Association, Berkeley, CA, USA, 2012.
- [5] Jeffrey Bickford, H. Andrs Lagar-Cavilla, Alexander Varshavsky, Vinod Ganapathy, and Liviu Iftode. "*Security versus energy tradeoffs in host-based mobile malware detection*", In Proceedings of the 9th international conference on Mobile systems, applications, and services (*MobiSys'11*). ACM, New York, NY, USA, 225-238, 2011.
- [6] Paul Brett, Mic Bowman, Jeff Sedayao, Robert Adams, Rob Knauerhase, and Aaron Klingaman. "*Securing the PlanetLab Distributed Testbed: How to Manage Security in an Environment with No Firewalls, with All Users Having Root, and No Direct Physical Control of Any System*", In Proceedings of the 18th USENIX conference on System administration (*LISA'04*). USENIX Association, Berkeley, CA, USA, 195-202, 2004.
- [7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "*Bigtable: a distributed storage system for structured data*", In Proceedings of the 7th symposium on Operating systems design and implementation (*OSDI'06*). USENIX Association, Berkeley, CA, USA, 205-218, 2006.
- [8] Georgios Chatzimilioudis, Andreas Konstantinidis, Christos Laoudias, and Demetrios Zeinalipour-

- Yazti. "Crowdsourcing with smartphones", In IEEE Internet Computing, Volume 16, 36-44, 2012.
- [9] Jeff Cohen, Thomas Repantis, Sean McDermott, Scott Smith, and Joel Wein. "Keeping track of 70,000+ servers: the akamai query system", In Proceedings of the 24th international conference on Large installation system administration (*LISA'10*). USENIX Association, Berkeley, CA, USA, 1-13, 2010.
- [10] Cory Cornelius, Apu Kapadia, David Kotz, Dan Peebles, Minh Shin, and Nikos Triandopoulos. "Anonymsense: privacy-aware people-centric sensing", In Proceedings of the 6th international conference on Mobile systems, applications, and services (*MobiSys'08*). ACM, New York, NY, USA, 211-224, 2008.
- [11] Geoff Coulson, Barry Porter, Ioannis Chatzigiannakis, Christos Koninis, Stefan Fischer, Dennis Pfisterer, Daniel Bimschas, Torsten Braun, Philipp Hurni, Markus Anwander, Gerald Wagenknecht, Sndor P. Fekete, Alexander Krller, and Tobias Baumgartner. "Flexible experimentation in wireless sensor networks", In Communications of the ACM, Volume 55, Issue 1, 82-90, 2012.
- [12] Eduardo Cuervo, Peter Gilbert, Bi Wu, and Landon Cox. "CrowdLab: An Architecture for Volunteer Mobile Testbeds", In Proceedings of the 3rd International Conference on Communication Systems and Networks (*COMSNETS'11*), IEEE Computer Society, Washington, DC, USA, 1-10, 2011.
- [13] Tathagata Das, Prashanth Mohan, Venkata N. Padmanabhan, Ramachandran Ramjee, and Asankhaya Sharma. "PRISM: platform for remote sensing using smartphones", In Proceedings of the 8th international conference on Mobile systems, applications, and services (*MobiSys'10*). ACM, New York, NY, USA, 63-76, 2010.
- [14] M. Dehus, and D. Grunwald. "STORM: simple tool for resource management", In Proceedings of the 22nd conference on Large installation system administration conference (*LISA'08*). USENIX Association, Berkeley, CA, USA, 109-119, 2008.
- [15] Jakob Eriksson, Lewis Girod, Bret Hull, Ryan Newton, Samuel Madden, and Hari Balakrishnan. "The pothole patrol: using a mobile sensor network for road surface monitoring", In Proceedings of the 6th international conference on Mobile systems, applications, and services (*MobiSys'08*). ACM, New York, NY, USA, 29-39, 2008.
- [16] Stavros Harizopoulos, and Spiros Papadimitriou. "A case for micro-cellstores: energy-efficient data management on recycled smartphones", In Proceedings of the Seventh International Workshop on Data Management on New Hardware (*DaMoN'11*), ACM, New York, NY, USA, 50-55, 2011.
- [17] David Johnson, Tim Stack, Russ Fish, Daniel Montalro Flickinger, Leigh Stoller, Robert Ricci, and Jay Lepreau. "Mobile Emulab: A Robotic Wireless and Sensor Network Testbed", In Proceedings of the 25th IEEE International Conference on Computer Communications (*INFOCOM'06*), IEEE Computer Society, Washington, DC, USA, 1-12, 2006.
- [18] Aman Kansal, Michel Goraczko, and Feng Zhao. "Building a sensor network of mobile phones", In Proceedings of the 6th international conference on Information processing in sensor networks (*IPSN'07*). ACM, New York, NY, USA, 547-548, 2007.
- [19] Keynote Systems Inc., Device Anywhere: <http://goo.gl/mCxft>
- [20] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. "Revisiting storage for smartphones", In Proceedings of the 10th USENIX conference on File and Storage Technologies (*FAST'12*). USENIX Association, Berkeley, CA, USA, 17-31, 2012.
- [21] Andreas Konstantinidis, Constantinos Costa, Georgios Larkou, and Demetrios Zeinalipour-Yazti. "Demo: a programming cloud of smartphones", In Proceedings of the 10th international conference on Mobile systems, applications, and services (*MobiSys'12*). ACM, New York, NY, USA, 465-466, 2012.
- [22] Andreas Konstantinidis, Demetrios Zeinalipour-Yazti, Panayiotis G. Andreou, Panos K. Chrysanthis, and George Samaras. "Intelligent Search in Social Communities of Smartphone Users", In Distributed and Parallel Databases, Volume 31, 115-149, 2013.
- [23] Emmanouil Koukoumidis, Li-Shiuan Peh, and Margaret Rose Martonosi. "SignalGuru: leveraging mobile phones for collaborative traffic signal schedule advisory", In Proceedings of the 9th international conference on Mobile systems, applications, and services (*MobiSys'11*). ACM, New York, NY, USA, 127-140, 2011.
- [24] Christos Laoudias, George Constantinou, Marios Constantinides, Silouanos Nicolaou, Demetrios Zeinalipour-Yazti, and Christos G. Panayiotou. "The Airplace Indoor Positioning Platform for Android

- Smartphones”, In Proceedings of the 13th IEEE International Conference on Mobile Data Management (MDM’12), IEEE Computer Society, Washington, DC, USA, 312-315, 2012.
- [25] Kaisen Lin, Aman Kansal, Dimitrios Lymberopoulos, and Feng Zhao. “Energy-accuracy trade-off for continuous mobile device location”, In Proceedings of the 8th international conference on Mobile systems, applications, and services (MobiSys’10). ACM, New York, NY, USA, 285-298, 2010.
- [26] Nagios Enterprises LLC., <http://www.nagios.org/>.
- [27] Rohan Narayana Murty, Geoffrey Mainland, Ian Rose, Atanu Roy Chowdhury, Abhimanyu Gosain, Josh Bers, and Matt Welsh. “CitySense: An Urban-Scale Wireless Sensor Network and Testbed”, In Proceedings of the 2008 IEEE Conference on Technologies for Homeland Security (HST’08), IEEE Computer Society, Washington, DC, USA, 583-588, 2008.
- [28] Adam J. Oliner, Anand P. Iyer, Eemil Lagerspetz, Ion Stoica and Sasu Tarkoma. “Carat: Collaborative Energy Bug Detection (poster and demo)”, 9th USENIX conference on Networked systems design & implementation (NSDI’12). USENIX Association, Berkeley, CA, USA, 2012.
- [29] Perfecto Mobile, <http://goo.gl/DS1P9>.
- [30] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. “A Blueprint for Introducing Disruptive Technology into the Internet”, A blueprint for introducing disruptive technology into the Internet. SIGCOMM Comput. Commun. Rev. 33, 1 (January 2003), 59-64, 2003.
- [31] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. “Paranoid Android: versatile protection for smartphones”, In Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC’10). ACM, New York, NY, USA, 347-356, 2010.
- [32] Jonathan Reams. “Extensible Monitoring with Nagios and Messaging Middleware”, In Proceedings of the 26th international conference on Large Installation System Administration: strategies, tools, and techniques (LISA’12). USENIX Association, Berkeley, CA, USA, 153-162, 2012.
- [33] Samsung, Remote Test Lab: <http://goo.gl/p7SNU>
- [34] Eric Sorenson and Strata Rose Chalup. “RedAlert: A Scalable System for Application Monitoring”, In Proceedings of the 13th USENIX conference on System administration (LISA’99). USENIX Association, Berkeley, CA, USA, 21-34, 1999.
- [35] Arvind Thiagarajan, Lenin Ravindranath, Katrina LaCurts, Samuel Madden, Hari Balakrishnan, Sivan Toledo, and Jakob Eriksson. “Vtrack: accurate, energy-aware road traffic delay estimation using mobile phones”, In Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys’09). ACM, New York, NY, USA, 85-98, 2009.
- [36] Tim Verry. “MegaDroid simulates network of 300,000 Android smartphones”, Extremetech.com, Oct 3, 2012. <http://goo.gl/jMaS8>.
- [37] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh. “MoteLab: a wireless sensor network testbed”, In Proceedings of the 4th international symposium on Information processing in sensor networks (IPSN’05). IEEE Press, Piscataway, NJ, USA, Article 68, 2005.
- [38] Demetrios Zeinalipour-Yazti, Christos Laoudias, Costantinos Costa, Michalis Vlachos, Maria I. Andreou, and Dimitrios Gunopulos. “Crowdsourced Trajectory Similarity with Smartphones”, IEEE Trans. on Knowl. and Data Eng. 25, 6 (June 2013), 1240-1253, 2013.
- [39] Yu Zheng, Lizhu Zhang, Xing Xie, and Wei-Ying Ma. “Mining interesting locations and travel sequences from GPS trajectories”, In Proceedings of the 18th international conference on World wide web (WWW’09). ACM, New York, NY, USA, 791-800, 2009.

YinzCam: Experiences with In-Venue Mobile Video and Replays

Nathan D. Mickulicz, Priya Narasimhan, Rajeev Gandhi
YinzCam, Inc. / Carnegie Mellon University
Pittsburgh, PA

nathan@yinzcam.com, priya@yinzcam.com, rgandhi@yinzcam.com
nmickuli@andrew.cmu.edu, priya@cs.cmu.edu, rgandhi@ece.cmu.edu

Abstract

YinzCam allows sport fans inside NFL/NHL/NBA venues to enjoy replays and live-camera angles from different perspectives, on their smartphones. We describe the evolution of the system infrastructure, starting from the initial installation in 2010 at one venue, to its use across a dozen venues today. We address the challenges of scaling the system through a combination of techniques, including distributed monitoring, remote administration, and automated replay-generation. In particular, we take an in-depth look at our unique automated replay-generation, including the dashboard, the remote management, the remote administration, and the resulting efficiency, using data from a 2013 NBA Playoffs game.

1 Introduction

Fans inside sporting venues are often far removed from the action. They have to rely on the large stadium video-boards to see close-ups of the action or be fortunate enough to be close to the field. YinzCam started as a Carnegie Mellon research project in 2008, with the goal of providing live streaming camera angles and instant hockey replays on their Wi-Fi-enabled smartphones inside the Pittsburgh Penguins' venue, Mellon Arena [5]. The original concept consisted of the Penguins Mobile app that fans could download and use on their smartphones, exclusively over the in-arena Wi-Fi network, in order to receive the unique in-stadium video content. Figure 1 shows the user experience at our initial installation for the Pittsburgh Penguins.

While YinzCam started with an in-stadium-only mobile experience, once the research project moved to commercialization, the resulting company, YinzCam, Inc. [21], decided to expand its focus beyond the in-venue (small) market to encompass the out-of-venue (large) market.

YinzCam is currently a cloud-hosted mobile-video service that provides sports fans with real-time scores, news, photos, statistics, live radio, streaming video, etc., on their mobile devices anytime, anywhere, along with live video and replays from different camera angles when these fans are inside sports venues. The game-time live video and instant replays are effectively gated to the stadium because of



Figure 1: Fan experience at the original Mellon Arena deployment for YinzCam, May 2010, showing three different smartphones displaying three different live camera angles for a Pittsburgh Penguins home game.

broadcast rights; all of the other content is available to fans outside the stadium. YinzCam's primary infrastructure is currently hosted on Amazon Web Services (AWS) [1] and supports over 7 million downloads of the official iOS, Android, BlackBerry, and Windows Phone mobile apps of 40+ NHL/NFL/NBA sports teams and venues within the United States. Twelve of these venues additionally support YinzCam's live video (including the NFL RedZone Channel at NFL stadiums) and instant replay technology over both Wi-Fi and cellular distributed antenna systems (DAS) [6].

The production of live video and instant replays requires the orchestration of both computerized systems and human efforts. In the video-processing plane, we utilize hundreds of virtual and physical machines that communicate over various wireless and wired networks. These systems include audio and video encoders, stream routers, web servers, media servers, and databases. In the management plane, we utilize remote-access technologies, automated-monitoring systems, and a game-day-operations team. We group these components into three major systems: the content-generation system (CGS), the mobile-wireless network (MWN), and the management-operations system (MOS).

Our first in-venue installation was located at Mellon Arena, the home of the Pittsburgh Penguins until 2010. In

this installation, the CGS, MWN, and MOS were all co-located in a small room that was adjacent to the arena's video production studio. Our equipment included a video encoder, three rack-mounted servers, and a terminal for the operator.

As we grew from one team in 2010 to five teams in 2011, we struggled to scale the initial design of our system to multiple installations with thousands of users. The load on our systems became too much for a single server or encoder to handle. Teams wanted to use more than the maximum of four camera angles that our initial designs allowed. New technologies like 4G LTE [17] and WiMAX [19] made cellular networks a viable option for the MWN, but our system was designed only for Wi-Fi. Replay cutting was a slow, labor-intensive, and error-prone process that frequently frustrated our system operators. Adding to the frustration, our operators were immersed in a high-pressure, game-time environment filled with distractions. Travel costs piled up while we shuttled operators back and forth across the country for every sporting event. There was a critical need for a cost-effective, low-latency scalable automated solution that addressed these challenges.

Contributions. The overarching goal of this paper is to describe how we evolved our initial system, with all of its limitations, into the scalable and efficient system that we use today to support thousands of sports fans across venues in the United States. Instead of simply scrapping our existing design and starting over, we slowly evolved our system by addressing each of the aforementioned challenges individually. In this paper, we discuss candidly the weaknesses in our original system architecture, the changes we introduced to make our system work efficiently and cost-effectively at scale, and the lessons we learned during this transformation. Concretely, our contributions in this paper are:

- A case-study of our migration to the cloud, including a discussion of how we chose which components to migrate and how the cloud helped us scale;
- A description and evaluation of our automated-replay-cutting system, including a discussion of how we improved its efficiency, reliability, and maintainability through automation, decoupling of subsystems, and configuration management;
- A description of our remote-management infrastructure, including our automated-monitoring systems and operations dashboards.

To the best of our knowledge, there exists no other automated mobile-replay generation and management system for sports venues.

The rest of this paper is organized as follows. Section 2 describes our mobile applications in detail and also gives a general description of the systems that support them. Section 3 describes our initial design for in-venue installations. Section 4 describes the technical challenges we faced in scaling up our initial design. Section 5 describes the changes we made to our initial design to solve these technical challenges,

and provides insight into the system we use today. Finally, section 6 discusses the lessons we learned throughout this process.

2 The YinzCam System

YinzCam develops the official mobile apps for more than 40 professional sports teams across four leagues (the NHL, NFL, NBA, and MLS). We provide these apps for over 7 million users across iPhone, iPad, Android, BlackBerry, and Windows Phone 7/8. YinzCam-powered apps have access to a wide variety of sports-related content such as news, photos, videos, podcasts, schedules, rosters, stats, and scores. This content is updated in real time as teams publish new statistics and media.

User Experience. YinzCam apps also support unique features not available in other sports apps, such as exclusive live-video streaming and on-demand instant-replays. From their seats or anywhere in the building, fans can use YinzCam apps to watch live game action from multiple camera angles on their smartphone. In addition, fans can watch replays of game events from the same set of camera angles. Some venues also provide fans with streams of popular sports-related TV-channels, such as the NFL RedZone channel [13].

To use our in-venue features, the user first configures his or her smartphone to connect to the in-venue Wi-Fi network (this step is not required for DAS networks). Next, after opening the app, the user navigates to a special in-venue section of the app. As shown in Figure 2, the user can then choose to watch live camera angles or browse through several replays of game action. Both live videos and instant replays are offered from multiple different camera angles. Only one camera angle may be viewed at a time.

These features are only accessible to fans while inside the team's home venue. Once inside and connected to an in-venue network, fans are able to access a special in-venue section of the mobile app. This section initially presents fans with a list of game highlights, as well as options for viewing additional replays or live streams (see Figure 2). Once a fan chooses a live stream or instant replay, he or she is presented with a selection of camera angles available for that item. Upon choosing an angle, the app begins playback of the corresponding video.

Although only a part of a large app that contains a wealth of other features and content, our in-venue live streaming and instant replays have proven to be the most technically-sophisticated (and technically-challenging) of all of the services we provide. We separate our system into a content plane and a management plane, with some components existing in both planes. The content plane includes components for encoding combined audio-video streams, converting streaming formats as needed by mobile devices, cutting segments of live video into instant replays, and serving live video and instant replays to clients over wireless networks. The management plane includes remote-access ser-



Figure 2: A user's navigation through the United Center Mobile application, in order to watch a replay.

VICES, monitoring systems, software configuration and deployment, and operations dashboards.

The content plane contains the content-generation system (CGS), which is a multi-stage pipeline that produces our live video and instant replays. This content is generated in three main stages. In the first stage, one or more YinzCam servers receive signals (i.e. feeds) from in-venue video-cameras and TV-tuners. The servers receiving these feeds encode them into compressed, high-quality data-streams suitable for transmission via IP networks. In the second stage, both the live streaming and the replay cutting components receive identical copies of these streams. The live streaming component converts the high-quality stream into a lower-quality stream suitable for playback over bandwidth-constrained wireless networks. The replay cutting component segments the live video stream into discrete video files, which are also encoded at a lower quality. Finally, in the third stage, both the live streaming and replay cutting services publish their content for consumption by mobile apps via the in-venue mobile wireless network (MWN).

Although just one stage in our pipeline, the final publishing and distribution step is unusually complex due to the wide variety of stream formats we need to support. Each smartphone platform that we support accepts a different set of stream formats. For example, iOS devices and Android 4+ accept Apple's HTTP Live Streaming (HLS) format [3]; both Android 2+ and BlackBerry accept the RTSP/RTP format [16]; and Windows Phone only accepts Microsoft's IIS Live Streaming (ISM) format [22]. To successfully stream to all of the platforms we support, we must provide each of our streams in the RTSP/RTP, HLS, and ISM formats.

The management plane contains services that our content-plane systems expose for the purposes of automated and manual management. Every YinzCam server provides remote access in the form of SSH or remote desktop (RDP) [7] access. Furthermore, some components like the replay cutter have human interfaces for monitoring and manual intervention. In later iterations of our system, this plane also includes automated-monitoring systems and virtual-private networks (VPNs).

The replay-cutting interface is the most complex component of our management system due to its unique requirements. To cut a replay, the operator first finds the correct segment of video in the each of the live-video streams. Once identified, the operator submits these segments for encoding, creating multiple video-clips (one for each stream) that are packaged in MPEG-4 containers [12]. The operator then annotates the replay with a title and description and publishes it to the app, which is a process where the system writes all of the metadata into a database table. Done manually, the entire process for a single replay may take 30 seconds or longer, and an operator will likely process over 150 replays during a typical game. The operator also has control over which replays are shown in the highlights list and can remove replays altogether if needed.

3 2010: Supporting our First Venue

While still a research project at Carnegie Mellon, YinzCam set up its first in-venue installation at Mellon Arena, the home of the Pittsburgh Penguins NHL team in 2008. Our base of operations was adjacent to the arena's video-production studio, which made it easy for us to obtain the our input video feeds. Our back-end equipment included a hardware encoder and three rack-mounted servers, all interconnected via the arena's single wired-network. The Wi-Fi network also shared this wired network for access to our in-venue services as well as the Internet. In addition to setting up our own back-end equipment, we also had the privilege of installing the Wi-Fi network at the arena.

In our first installation, the CGS pipeline consisted of an encoder that provided two outputs, one to a live-streaming server and the other to a replay-cutting server. Figure 3 is a diagram of this architecture. This Figure also shows the terminal that our operators used to monitor each of the system components via RDP [7].

The video-capture component converted standard-definition, analog video into compressed streams suitable for transmission over IP networks. For this component, we used an Axis Q-series video encoder [4], which converted up to 4 analog, NTSC-formatted inputs into 4 digital,

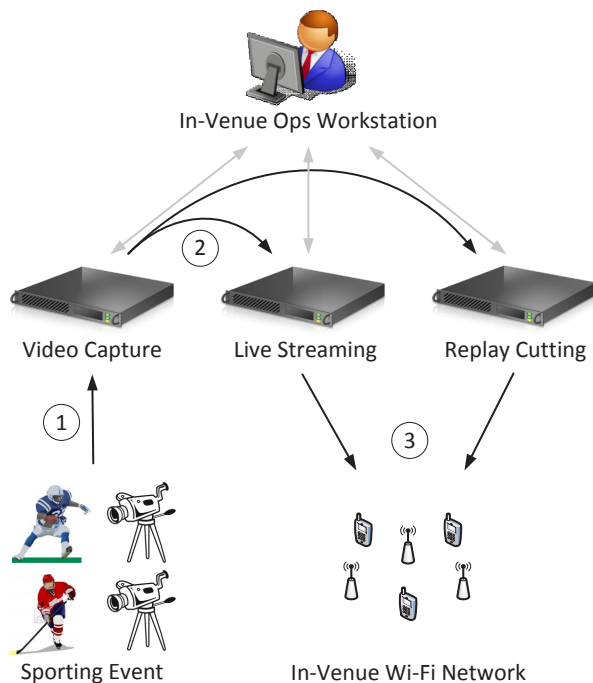


Figure 3: The architecture diagram of the first YinzCam installation at Mellon Arena. Black lines indicate content flows, while gray lines indicate management traffic (2010).

H.264-encoded streams. The encoder served these streams via the stadium’s wired network segment in two formats, Motion-JPEG (MJPEG) [11] and RTSP/RTP [16].

Because the Axis encoder lacked the resources to handle hundreds of simultaneous connections, we wrote a custom proxy for the encoder’s MJPEG output that could run on general-purpose hardware. This proxy server was a .NET application hosted on a Dell R200 server running the Windows Server 2008 operating system. Our proxy server received and buffered MJPEG frames from the Axis encoder, and it also maintained HTTP connections to each mobile app displaying a live stream. When the proxy received a new MJPEG frame from the encoder, it immediately sent a copy of the frame to each connected client. This decoupled the clients from the Axis encoder, allowing us to scale the number of clients by scaling our server capacity.

We also used custom software to cut replays from the live streams. Like the live-stream proxy, this software also received and buffered MJPEG frames from the encoder. However, instead of distributing frames to clients, the replay cutter buffered the last 5 minutes of frames in memory. The replay cutter indexed these frames by the time when they were received, with a separate index for each stream. To cut a replay, the system first used these indexes to look up the sequence of frames between the start and end times for the replay. The replay cutter then spawned encoding processes to concatenate the sequences, one for each combination of

live stream and mobile platform. This process created several MP4 [12] files that were made available for download over the in-venue Wi-Fi network via an IIS web server running on the same machine.

The Wi-Fi network initially used 4 Xirrus access points [20] covering 2000 season-ticket-holder seats. This was later expanded to 8 access points to meet increased demand. Each access point had 8 radios. Of the 8 radios, 3 use the 2.4 GHz band and 3 use the 5 GHz band. One or two radios are used as monitors, providing feedback about network conditions.

In the initial installation, all of the monitoring, configuration, and software maintenance were entirely manual processes managed by a single system-operator. During a game, the operator’s primary task was to find and annotate each replay. If any problems occurred, the operator was expected to diagnose and resolve the problem using remote desktop access to each of the other servers.

The replay cutter provided a Windows-Forms [8] interface for managing its operation. This interface allowed the operator to seek backwards through the last 5 minutes of live video to find replays. Once the operator identified the start and end positions for the replay, the interface communicated the corresponding timestamps to the replay cutter for encoding. Once encoded, the operator assigned a title and description to the replay and published the replay into the app.

4 2011: Scaling to Four Venues

In 2011, YinzCam expanded to work with its first three NFL teams: the New England Patriots, the Pittsburgh Steelers, and the San Francisco 49ers. We built nearly-identical copies of our Mellon Arena installation at their respective home stadiums, Gillette Stadium, Heinz Field, and Candlestick Park. We sent an operator to each stadium for every home game during the 2011 season. During the game, the operator managed the replay cutter and responded to any on-site problems.

Although technically feasible, scaling up to these 4 new NFL installations caused significant management problems. While the original system architect could be on-site (or at least on-call) for every Penguins game, NFL games are often played simultaneously, so we had to send less-experienced operators to some games. This made it difficult for us to diagnose and resolve problems that occurred in the field. Travel costs for our operators were also a concern.

We first attempted to resolve these issue by extending the initial design with remote access. As shown in Figure 4, we added a dedicated server that hosted a VPN service, allowing remote access to the network from any Internet-accessible location. This VPN provided all of our operators with remote access to our systems. This lessened the burden on the single operator who managed the replay cutter, while simultaneously giving remote systems-experts the ability to troubleshoot problems.

Because of its maturity and low operation-cost, we chose OpenVPN [14] to provide a secure connection between re-

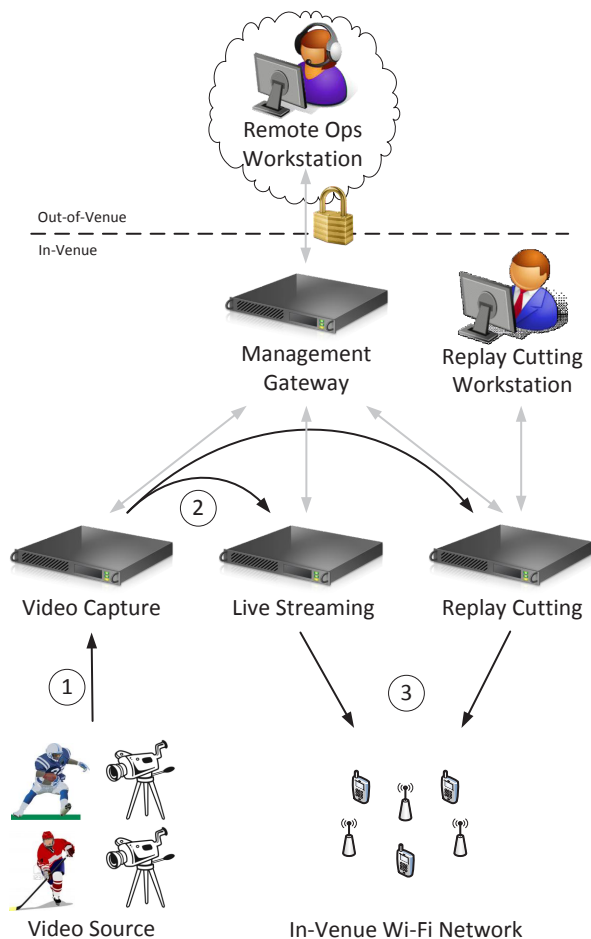


Figure 4: The revised architecture diagram of our initial design, showing the infrastructure added for management. Black lines indicate content flows, and gray lines indicate management traffic. The connection between the Remote Ops Workstation and the Management Gateway is secured by a VPN (2011).

mote system-operators and our installations. A dedicated management-server at each installation hosted the OpenVPN service. Our system operators installed OpenVPN clients on their work machines, which gave them secure network-access to all of our installations anywhere at any time. This connection allowed our operators to open tunneled SSH and RDP [7] connections to each of our servers and manage them as if they were located on-site.

While this was sufficient for the 2011 season, we continued to encounter management issues as we expanded to 12 teams during the 2012 season. Monitoring, configuration, and software maintenance was still an entirely-manual process. Although manageable with just 4 installations, as we scaled up, major system-changes and software upgrades became tedious and error-prone tasks. Furthermore, the lack of a single network that connected all of our installations made

it difficult for us to deploy infrastructure-wide management or monitoring software.

On the business side, the cost of having a person on site for every game was tremendous. We had to purchase airfare, meals, ground transportation, and lodging for each operator sent to a game not played in Pittsburgh. For example, during the 2011 49ers season, we purchased 10 round-trip flights from Pittsburgh to San Francisco. These costs were unbearable for a young company like ours.

In addition to management issues, replay cutting for NFL games has much more strenuous operations-requirements than for NHL games. While the replays published to NHL apps are chosen at the operator's discretion, our NFL apps show replays of every play of every game. Although this makes it easier for the operator to find replays, the operator has to be focused on cutting replays during every second of the game. Furthermore, NFL replays cannot be skipped, meaning that the operator's replay queue frequently became backlogged. Not only did this make replay cutting more stressful, but some replays also took minutes to generate due to the backlog. We clearly needed a more efficient system.

Finally, our system was not prepared for the introduction of 4G cellular-technologies in late 2011. These technologies, coupled with the distributed-antenna system (DAS) [6], allowed cellular networks to provide high-bandwidth, low-latency data-connections in high-user-density environments. Such networks were ideal for our video-streaming application in stadiums or arenas. However, since our initial design assumed that the MWN would always be Wi-Fi network, it required a private LAN connection between the wireless network and our servers. Cellular networks do not permit such connections, and all data services for these networks must operate on the public Internet. Due to bandwidth constraints, using the venue's Internet connection to meet this requirement was also out of the question.

5 2012: Our Infrastructure Today

Today, YinzCam operates 12 in-venue installations in production across 12 NFL teams, 1 NHL team, and 1 NBA team. We no longer struggle to support new installations or handle heavy game-day-usage, and we can easily support multiple MWNs co-existing in a single installation. To achieve success at this scale, we redesigned our system to be more flexible, modular, and to embrace new technologies in the areas of cloud computing, elastic scaling, system automation, and distributed monitoring.

5.1 To The Cloud

This section describes, in detail, our transition from an architecture relying entirely on private infrastructure to one heavily-dependent upon Amazon's public EC2 cloud [2]. The transition was neither smooth nor seamless, and we candidly discuss the rationale for each of our design changes. We also discuss, in hindsight, the advantages and disadvantages of our approaches.

5.1.1 A Global VPN

We first turned to cloud computing to solve the problem of needing an infrastructure-wide private-network with which to coordinate the management all of our installations. One of the challenges we faced during the 2011 season was the tedium of manually managing each deployment as a separate, unique installation. We needed a way to centrally manage the configuration of each system component across all of our installations.

In pursuit of this goal, we knew that we needed a single, common network across all of our installation. We reasoned that hosting this component on the cloud would result in a more reliable service than hosting it at our offices on a cable-broadband connection. It would also be cheaper, since we would not need to purchase or operate the infrastructure ourselves. In late 2011 we set up a low-capacity, Linux-based EC2-instance to act as an OpenVPN server. We migrated all client certificates to this server, and converted the OpenVPN servers running on each installation's management gateway into a client of our new, cloud-hosted VPN.

One problem we faced while setting up this VPN was addressing. While it would be simple to assign blocks of IP addresses from a reserved, private address-space, we had to be careful to avoid address conflicts. Both sporting venues and Amazon EC2 use blocks of addresses chosen from all of the reserved private IP address blocks [15] (192.168.0.0/16, 172.16.0.0/12, and 10.0.0.0/8), with most heavy usage in the 192.168 and 10 blocks. We chose at random a /16 prefix in the 10-block, 10.242.0.0/16, and we hoped that this space never conflicted with any other routable network inside of a venue or on Amazon EC2. In hindsight, this was poor choice because our 10.242 subnet did conflict with the private addresses of some of our EC2 instances. We resolved these conflicts using host routes, but this complicated our server configuration.

We also developed a hierarchical strategy for assigning our global VPN's address space. We delegated a /24 subnet to each installation, starting with 10.242.1.0/24 for our Pittsburgh office and incrementing the third octet for each installation. We assigned the final octet of the address to each machine based on its role; for example, the VPN interface for all live streaming servers is assigned the address 10.242.xxx.20. We grouped these per-role addresses into blocks of 10, so that we could assign sequential addresses for up to 10 machines in the same role.

While our cloud-based VPN worked well normally, it had trouble coping with failures. The main weakness of our initial design was the single point of failure on the cloud. Should this server fail, the entire VPN would disappear. We reasoned that the use of a low-TTL DNS record and an EC2 Elastic IP Address would allow us to quickly boot up and route traffic to a new VPN server should any problems arise.

However, we found that OpenVPN's automatic reconnection mechanism was not reliable in the case of network or server failure. During an unexpected failure, some clients

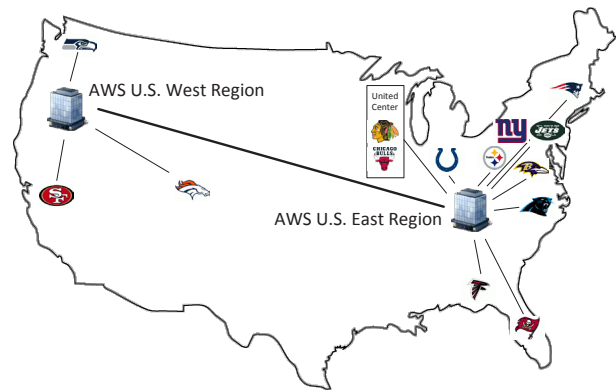


Figure 5: YinzCam's global VPN, showing the 12 installations that we remotely administer.

fail to detect a connection loss. When the server comes back online, these clients do not attempt reconnection and remain permanently offline. The same situation can occur after a network outage. Getting these systems back online requires an on-site technician to either reboot the server or log in and restart the VPN client. Due to these issues, we decided that our VPN was not reliable enough for continued production usage, and began examining alternatives.

Today, we use peer-to-peer (P2P) VPN software to solve our reliability problems. Instead of OpenVPN's client-server model, P2P VPN systems view each machine as a node in a mesh network. These systems use dynamic-routing protocols like OSPF [9] to find the best available routes for traffic. Like the Internet, as long as a sequence of connected VPN nodes exists between any two points, traffic can flow between them. P2P VPNs also permit two nodes to form direct connections if possible, eliminating unnecessary routing hops.

In the cloud, we now have 6 EC2 instances across 2 AWS regions that serve as rendezvous points for VPN connections. Each of our machines is configured with the FQDNs of the 3 VPN rendezvous instances in the nearest AWS region. Upon startup, the VPN software on the machine connects to all three of these nodes. As long as at least 1 of the 3 nodes is functioning, the node has full access to the VPN. Between the 2 regions, the 3 nodes in each region maintain connections to the three nodes in the other region, providing connectivity between the regions. Figure 5 shows how these connections are organized across our U.S. installations.

5.1.2 Supporting Multiple MWNs

The second phase of our cloud migration occurred in response to the widespread adoption of 4G technologies (LTE and WiMAX) by mid-2012. With the installation of DAS systems in several sporting venues, cellular networks could provide the high-bandwidth, low-latency data-connections required for our in-venue technologies. Unfortunately, cellular networks do not offer the same direct, private-LAN con-

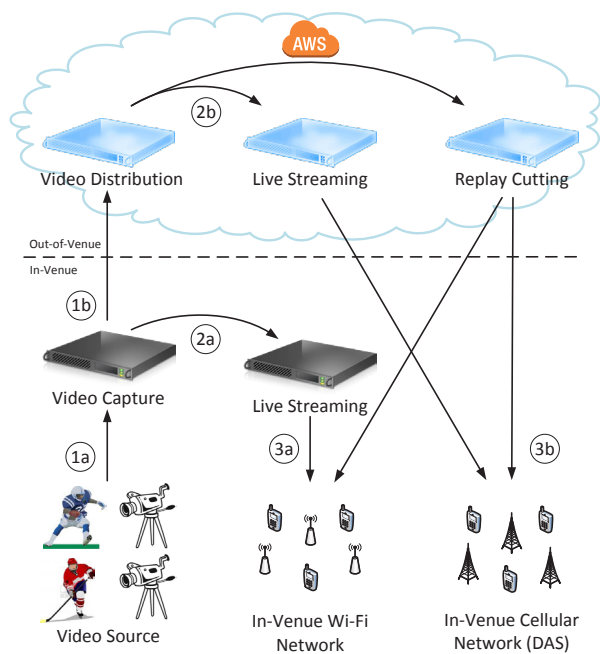


Figure 6: Our Wi-Fi/DAS hybrid architecture, utilizing two parallel content-generation pipelines. For clarity, management systems are not shown (2012).

nections as in-venue Wi-Fi networks; instead, all external traffic must come from the Internet.

Our first approach was to provide external access to our in-venue servers via the stadium’s Internet connection. Unfortunately, supporting just 1000 concurrent connections to our live-video service with 400-kbps streams requires 400 Mbps of bandwidth. Stadiums typically have Internet connection capacities of 100 Mbps to 1 Gbps, shared across all services in the building (including Wi-Fi Internet, ticketing, vending, etc.). Reserving such a large portion of this connection for our use was simply not feasible.

We turned to cloud computing to provide the needed bandwidth. By sending a third copy of the video encoder’s output to the cloud, we were able to duplicate our entire content-generation pipeline on Amazon EC2. The live streams and instant replays generated by this pipeline were superficially-equivalent to those generated by the in-venue pipeline. However, the cloud pipeline’s output would already be on the cloud, ready for distribution to 4G clients using the vast Internet-bandwidth available to us through AWS.

The architecture we designed to implement our idea is shown in Figure 6. We sent one copy of each of the source video-streams to the cloud at 2 Mbps average bit-rate each. For a typical installation processing 4 simultaneous streams, this required a tolerable 8 Mbps of upload bandwidth from the venue. We configured virtual machines on EC2 in the same way as our in-venue physical live-streaming servers and replay cutters. Instead of receiving feeds directly from the venues, we set up these virtual machines to retrieve video streams from a video-distribution server, using software to

proxy the streams received from the video-capture server in the venue.

5.2 Automated Replay Cutting

As mentioned earlier, replay cutting prior to automation was by far the most labor-intensive and error-prone task in all of our system operations. During a typical three-hour game, manual replay-cutting required the complete and undivided attention of a single system-operator during every second of game time. This system was subject to numerous sources of delays and errors, stemming from the time taken to find replay boundaries and the unpredictable rate at which new replays needed to be cut.

Driven by our frustration with manual replay-cutting, we were able to automate the entire process with production-quality accuracy for football, hockey, and basketball games. Furthermore, we improved our system’s robustness by refactoring its components into three separate processes that communicate via middleware, allowing each component to fail independently. We also improved maintainability by introducing automated configuration management and upgrade software.

Before beginning our implementation, we needed to find an approach for automating our system. Our replay cutter operates by buffering short segments of video, indexing these segments by time, and then joining the segments between two timestamps into a video file. We wanted automate the selection of timestamps to eliminate the burden of searching manually through video streams to find the start and end points of replays. We developed a set of rules that estimate the duration of replays. For example, in football, plays that advance the ball further generally take longer to execute than short-distance plays. We slowly improved these rules, using the manual replays from previous games as training data, until our system was able to consistently estimate correct replay durations.

The timing rules provided us with all of the information needed to automate the replay cutting process. In our automated system, the video-buffering and encoding component remains the same as in the manual system. We added a new component that executes our timing rules for each replay, outputting start and end timestamps. A manager component tracks all of the replays and triggers the video cutter when start or end timestamps change. Once a replay is cut, the manager inserts the title, description, and video-file URLs into the list of available replays. A web server sends this list to our apps as an XML document [18] whenever the user opens the instant replay feature of our apps.

Also, there are many reasons that the automatic cut may be incorrect and require manual intervention. Data entry errors occasionally occur. In rare cases, our automated estimation is incorrect and needs adjustment. To handle these cases, we provide the option for an operator to override the automatic cut of any replay.

Although some rare cases still require human intervention, replay cutting now requires much less human effort than in

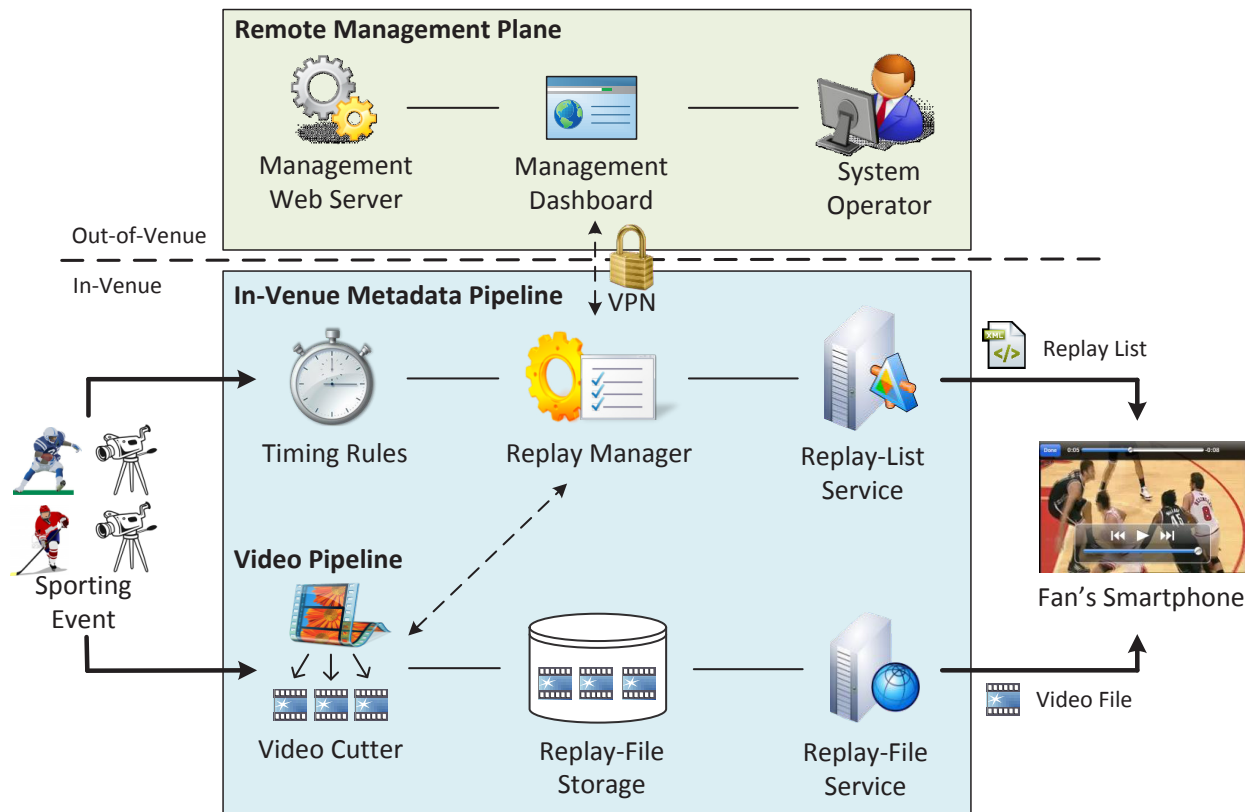


Figure 7: The architecture of our automated replay cutter, showing the in-venue and out-of-venue components.

the entirely-manual system. Operators now take a monitoring role, watching replays and correcting errors as needed. The reduction in human effort has made our replay cutter more efficient, reducing the time taken to generate our replays from 30 seconds (or more) to consistently less than 10.

We also introduced an automated configuration management and deployment system to ease maintenance of our software across multiple deployments. The core of this system is a central database of all currently-running components, their configurations, and their software versions. When upgrading a software component, the administrator simply has to upload the appropriate software artifact (usually a Java JAR file) to the system and select which installations to update. The automated deployment system handles shutting down the component to upgrade, copying the software update, and restarting the component. Configuration changes (in the form of XML documents) are also uploaded to the server and deployed using a similar stop-copy-restart process.

5.3 Remote Management Infrastructure

To keep our systems running smoothly, we need to quickly identify and respond to any problems that occur. These problems range from minor errors, such as video-encoding mistakes or replays containing the wrong video, to complete

outages of machines or even entire installations. This wide range of problems requires an equally wide range of monitoring and management systems.

5.3.1 Distributed Automated Monitoring

We use a distributed approach to monitor our infrastructure, where each machine individually monitors its own components and alerts operators of any problems. While a centralized approach would permit more advanced analysis, it would also require the transmission, storage, and processing of large quantities of system-log data. We opted for a distributed approach to avoid these performance and cost obstacles.

Our most basic level of monitoring is based on checking system-level metrics for anomalies. A daemon process on each machine monitors resources for excessive usage; for example, high CPU utilization, memory exhaustion, low disk space, and high network utilization. This daemon also monitors our configuration files for unexpected changes, which helps us catch problems early and provides auditing and other security benefits. When anomalies are detected, these daemons alert the entire operations team via email with details of the problem.

In addition to system-level monitoring, we also use service-log monitoring. A daemon on each machine mon-

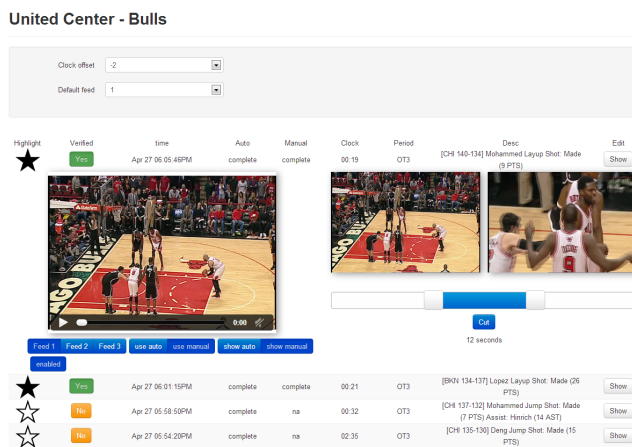


Figure 8: Operator perspective: A screenshot of our replay-cutting dashboard in action. The game shown is the NBA Playoffs game between the Chicago Bulls and the Brooklyn Nets on Saturday, April 27, 2013.

itors logs from both standard network services, like sshd, as well as in-house services like our video encoders and replay cutters. This daemon uses rules specific to the log format to detect unusual log messages (e.g. errors, repeated warnings, etc.), which are then emailed to our operations team for further review.

Our final tier of automated monitoring performs periodic liveness checks. The daemons in this tier periodically make network requests to ensure that our services are functioning normally. The simplest of these tests is a ping test to ensure network connectivity. We also perform more detailed service-level monitoring. For HTTP-based services, the monitoring daemons issue HTTP requests and check the responses for correctness. For example, to monitor our live streaming for iOS and Android 4 devices, our monitoring system periodically issues HTTP requests for the HLS playlist file and ensures that the response is valid. It then makes additional HTTP requests to ensure that the video files in the playlist are actually available.

5.3.2 End-to-End Monitoring

In addition to our automated monitoring, we also use tools for end-to-end monitoring. We use modified smartphones which allow our system operators to use the in-venue features of our apps as if they were physically located inside of the stadium or arena. We accomplished this by equipping smartphones with VPN connections, and using a special version of our apps that play live video and instant replays using the VPN connection.

We've found that end-to-end monitoring is necessary to ensure that the fan experience is not compromised due to system failure. Without end-to-end monitoring, it is very difficult to ensure that fans are seeing the correct app content. Furthermore, this monitoring allows us to do manual checks of our video content for problems that are difficult to detect manually, such as video distortion and encoding errors.



Figure 9: User perspective: A photo of the United Center Mobile app in action during a Chicago Bulls basketball game. Using his smartphone, this fan is able to watch the game close-up from his upper-level seat.

5.3.3 Replay-Cutter Dashboards

We have built various web services and web-based dashboards for managing the more complex, distributed components of our system, such as our replay cutter. We have two separate management dashboards for this system. The first is a systems-level dashboard, which allows us to remotely monitor, configure, and upgrade all of the replay cutters in our fleet. The second is our operator dashboard, which gives operators the ability to view and edit replays as well as select which replays should be shown in our apps as highlights.

Our systems dashboard ties in to the middleware layer of our replay cutter, allowing system administrators to monitor and manage all of the components of each replay cutter in our fleet. Once a replay-cutter's components are linked with the system dashboard, administrators can manage each of its components independently. The dashboard shows the current state of each component and provides commands to stop and start the execution of each. To aid in debugging, administrators can also view the logs generated by each component. This dashboard also provides the administrative interface to our configuration management system, which our system administrators use to update our software and system configurations as mentioned earlier.

Our replay cutting system also provides an interface that allows our operators to manually override the decisions made by the automated system. This interface takes the form of a web-based dashboard. Using this dashboard, the operator may adjust the time synchronization between the events feed and the video streams, manually override the start and end times for any event, and even disable replays entirely for certain events. The dashboard also provides insight into the internal operation of the system, such as the list of events and the systems progress in cutting each replay. Finally, the dashboard allows the operator to view the automatically- and

manually-cut replay videos for every game event, camera angle, and mobile platform.

This combination of features allows replay-cutter operators to primarily act as an observer to the system. Typically, operators just watch the automatically-cut replay videos via the dashboard and check for correctness. When the system creates an incorrect replay, our dashboard gives the operator the ability to easily intervene in the automatic operation. The streamlined interface also allows a single manager to easily monitor the replay cutting system across multiple concurrent sporting events.

5.3.4 Replay-Cutting Efficiency

By timing our system's progress through the various stages of replay cutting, we were able to compare the efficiencies of the manual and automated versions of our system. Our metric of interest is "time-to-app" (TTA), which is the time duration measured between our system receiving a new game event and the replay of that event appearing in the app for fans to watch. We measured the TTA for replays using both automated and manual workflows during the Chicago Bulls NBA Playoff game on April 27, 2013.

We measured the TTA of each workflow by summing the time spent in individual stages of the system. The stages we measured were search, encoding, and publishing. We used our replay-cutter dashboard to measure both workloads by timing the delay between state transitions in the dashboard. For example, when a new event is received, the dashboard shows a new line for the event in the ENCODING state. Once encoding is complete, the state changes to PUBLISH, indicating that the replay is being published in the replay list. The time taken for the ENCODING state to change to PUBLISH is the encoding-stage duration. The total time for the event to transition from ENCODING to PUBLISH to COMPLETE is the TTA for the automated workflow. For the manual workflow, we recorded an additional phase called SEARCH, where a human operator manually searched through the video stream for the replay. The TTA for the manual workflow was the total time for the transition from SEARCH, through ENCODING and PUBLISH, and finally to COMPLETE.

The result of our experiment is shown in Figure 10. The figure shows the average TTA for both the manual and automated workflows during the game. Each bar is broken down by pipeline stage, showing where time was most spent on average.

The results show a significant difference in average TTA between the two workflows. The average TTA for the manual workflow was 34.30 seconds, while the average TTA was just 5.38 seconds for the automated workflow. By automating our workflow, we achieved an 84.3% reduction in average TTA. While encoding and publishing times remained nearly constant in both workflows, most of the time savings in the automated workflow resulted from the elimination of the manual-search step.

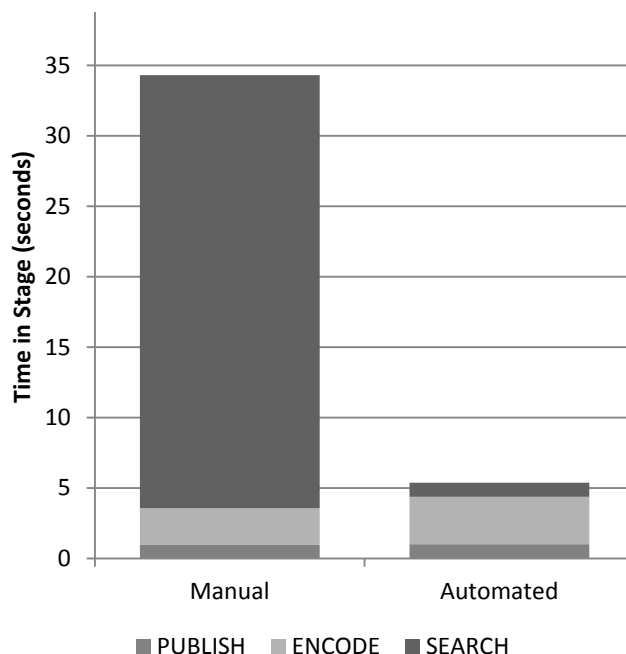


Figure 10: The time-to-app comparison of our manual and automated workflows during the NBA Playoffs game between the Chicago Bulls and the Brooklyn Nets on Saturday, April 27, 2013.

6 Lessons Learned

In this section, we summarize some of the lessons that we have learned in building and evolving our YinzCam in-venue infrastructure over the course of three years.

- **Don't locate operations efforts in high-pressure environments.** There should be some amount of physical separation between the operations team and the field environment. Intense, high-pressure environments are filled with distractions, which means that problem-solving takes longer and resources are harder to access. We accomplished this by migrating our operations efforts (both people and infrastructure) to a single remote location from where all of our venues can be managed simultaneously.
- **Take advantage of the cloud for short-term-CPU-intensive or bandwidth-intensive processes.** The cloud's pay-as-you-go model makes it well-suited for short-term tasks. Instead of purchasing physical servers that will mostly sit idle, we migrated these tasks to the cloud to lower our operations costs. Also, cloud-infrastructure providers have vast amounts of Internet bandwidth available for bandwidth-intensive services like video streaming. We leveraged Amazon EC2's bandwidth to provide our content to fans on cellular networks. While our out-of-stadium capabilities are outside the scope of this paper, our other work [10] also demonstrates how the cloud is an ideal platform for our usage profile (spiky loads on gamedays, mostly idle on non-game-days).

- **Ensure that system administrators have remote access to all systems.** The costs of sending technicians on-site for repairs can quickly add up. By allowing administrators to fix problems remotely, many of these costs can be eliminated. Our global VPN architecture allows us to monitor all of our physical assets remotely, providing insight into problems that need resolution. In addition, having a technical contact at remote sites also helps in case the system is not responding over the network.
- **Automate labor-intensive and error-prone manual efforts whenever possible.** Automating such tasks reduces human error and frees up staff to do other work. Furthermore, automated tasks are easily repeatable across multiple systems. Our automated replay-generation system is a clear case where we were able to efficiency gains and cost reduction through automation.
- **Use end-to-end monitoring of the user experience.** It is important to have insight not only into the system behavior, but also into the end-user's experience. We accomplished this by ensuring that our operations staff is equipped with smartphones that mimic the user experience inside the venue from a remote location. It should be noted that there are some problems cannot be reproduced in this manner, e.g., low Wi-Fi signal strength in seating locations.
- **Allow system operators to manually override automatic decisions.** No system is perfect; automated systems will eventually make mistakes due to unforeseen circumstances. Ensure that systems allow operators to manually override automatic decisions. Our replay-cutting dashboard allows human operators to compensate for incorrectly-cut replays.

While Yinzcam's architecture and internal details might be application specific, we believe that these lessons are broadly applicable to other systems. Specifically, our lessons apply to systems that require the remote monitoring of geographically distributed sites, the separation of services at the edge versus in remote data-centers, and the use of application-specific automation to increase efficiency and lower costs, wherever possible.

Acknowledgments

We would like to thank our shepherd, Patrick Cable, for helping us improve our paper. We would also like to thank Justin N. Beaver for his work in the development of our automated replay cutting system.

References

- [1] Amazon Web Services. Amazon Web Services. . URL <http://aws.amazon.com/>.
- [2] Amazon Web Services. Amazon Elastic Compute Cloud. . URL <http://aws.amazon.com/ec2/>.
- [3] Apple, Inc. HTTP Live Streaming. URL <https://developer.apple.com/resources/http-streaming/>.
- [4] Axis Communications AB. Axis Video Encoders. URL http://www.axis.com/products/video/video_server/index.htm.
- [5] ComputerWorld. Context on ice: Penguins fans get mobile extras. "http://www.computerworld.com/s/article/9134588/Context_on_ice_Penguins_fans_get_mobile_extras".
- [6] Crown Castle. DAS. URL <http://www.crowncastle.com/das/>.
- [7] Microsoft, Inc. Remote Desktop Protocol. . URL [http://msdn.microsoft.com/en-us/library/windows/desktop/aa383015\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa383015(v=vs.85).aspx).
- [8] Microsoft, Inc. Windows Forms. . URL <http://msdn.microsoft.com/en-us/library/dd30h2yb.aspx>.
- [9] J. Moy. OSPF Version 2. RFC 2328 (INTERNET STANDARD), Apr. 1998. URL <http://www.ietf.org/rfc/rfc2328.txt>. Updated by RFCs 5709, 6549, 6845, 6860.
- [10] N. Mickulicz, P. Narasimhan, and R. Gandhi. To Auto Scale or not to Auto Scale. In *Workshop on Management of Big Data Systems*, San Jose, CA, June 2013.
- [11] National Digital Information Infrastructure and Preservation Program. MJPEG (Motion JPEG) Video Codec. . URL <http://www.digitalpreservation.gov/formats/fdd/fdd000063.shtml>.
- [12] National Digital Information Infrastructure and Preservation Program. MPEG-4 File Format, Version 2. . URL <http://www.digitalpreservation.gov/formats/fdd/fdd000155.shtml>.
- [13] National Football League. NFL RedZone. URL <http://redzonetv.nfl.com/>.
- [14] OpenVPN Technologies, Inc. OpenVPN. URL <http://openvpn.net/>.
- [15] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918 (Best Current Practice), Feb. 1996. URL <http://www.ietf.org/rfc/rfc1918.txt>. Updated by RFC 6761.
- [16] H. Schulzrinne. Real time streaming protocol (RTSP). 1998.
- [17] S. Sesia, I. Toufik, and M. Baker. *LTE: The UMTS long term evolution*. Wiley Online Library, 2009.
- [18] The World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0 (Fifth Edition). URL <http://www.w3.org/TR/xml/>.
- [19] WiMAX Forum. WiMAX-Part I: A technical overview and performance evaluation. 2006.
- [20] Xirrus, Inc. Wireless Arrays & Access Points. URL <http://www.xirrus.com/Products/Wireless-Arrays.aspx>.
- [21] YinzCam, Inc. "http://www.yinzcam.com".
- [22] A. Zambelli. IIS smooth streaming technical overview. 2009.

Challenges to Error Diagnosis in Hadoop Ecosystems

Jim (Zhanwen) Li¹, Siyuan He², Liming Zhu^{1,3}, Xiwei Xu¹,
Min Fu³, Len Bass^{1,3}, Anna Iiu^{1,3}, An Binh Tran³

¹NICTA, Sydney, Australia

²Citibank, Toronto, Canada

³School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

Abstract

Deploying a large-scale distributed ecosystem such as HBase/Hadoop in the cloud is complicated and error-prone. Multiple layers of largely independently evolving software are deployed across distributed nodes on third party infrastructures. In addition to software incompatibility and typical misconfiguration within each layer, many subtle and hard to diagnose errors happen due to misconfigurations across layers and nodes. These errors are difficult to diagnose because of scattered log management and lack of ecosystem-awareness in many diagnosis tools and processes.

We report on some failure experiences in a real world deployment of HBase/Hadoop and propose some initial ideas for better trouble-shooting during deployment. We identify the following types of subtle errors and the corresponding challenges in trouble-shooting: 1) dealing with inconsistency among distributed logs, 2) distinguishing useful information from noisy logging, and 3) probabilistic determination of root causes.

1. Introduction

With the maturing of cloud and Hadoop technologies, more and more organizations are deploying and using systems in the Hadoop ecosystem for various purposes. Hadoop is an ecosystem that consists of multiple layers of largely independently evolving software and its deployment is across distributed nodes and different layers.

Even for experienced operational professionals with limited experience with the Hadoop ecosystem, the deployment and use is highly error-prone and error diagnosis and root cause identification takes a significant amount of time.

Traditionally, logs and error messages are important sources of information for error diagnosis. In a distributed system, logs are generated from multiple sources with different granularities and different syntax and semantics. Sophisticated techniques have been proposed to produce better logs or analyze existing logs to improve error diagnosis. However, there are a number of limitations of the existing approaches for the situation outlined above.

Consider one of the error messages that we encountered in

our experiments “java.net.ConnectException: Connection refused “. One existing approach is to correlate error messages with source code. Yet knowing where in the Java library this message was generated will not help determine the root cause. The cause of this error is, most likely, a misconfiguration but it is a misconfiguration that indicates inconsistency between multiple items in the ecosystem. Trouble shooting an error message such as this requires familiarity with the elements of the ecosystem and how they interact. This familiarity is primarily gained through experience, often painful. Furthermore, the messages leading up to this error message may be inconsistent or irrelevant. They are usually voluminous, however.

Providing assistance to non-expert installers of a complicated eco-system such as HBase/Hadoop is the goal of the work we report on here. In this paper, we report some failure experiences in real world deployments of HBase/Hadoop. Specifically, we focus on three key challenges: 1) dealing with inconsistency among distributed logs, 2) distinguishing useful information from noisy logging, and 3) probabilistic determination of root causes.

There are two assumptions about this work. First, it came out of observing and studying errors committed by non-expert installers of Hadoop ecosystems. Our target is system administrators and non-experts in HBase/Hadoop. Second, we assume that the developers of such systems will not change the way they record logs significantly although we do hope they produce them with operators more in mind. Thus our initial solutions are around dealing with inconsistency and uncertainties with existing logs. The case studies are all based on an Hadoop/HBase [3][5] cluster running on AWS EC2s[2].

The contributions of this paper include:

1. Identification of different types of errors in Hadoop ecosystem deployment using real world cases and investigations into the root causes of these errors. The majority of errors can be classified into four types:

- **Operational errors** such as missing/incorrect operations and missing artifacts. Errors introduced during restarting/shutting down nodes, artifacts (files and directories) not created, created with the wrong permission or mistakenly moved and disallowed operations due to inconsistent security environment are the major ones.

- **Configuration errors** include errors such as illegal, lexical, and syntax errors in standalone software systems and cross-systems/nodes inconsistency in an ecosystem.
- **Software errors** include compatibility issues among different parts of an ecosystem (e.g. HBase and HDFS compatibility issues) and bugs.
- **Resource errors** include resource unavailability or resource exhaustion, especially in cloud environment, that manifest themselves in highly uncertain ways and lead to system failures.

The diagnosis of these errors and locating the true causes is more difficult in an ecosystem setting, which leads to our second contribution.

2. Identified specific error diagnosis challenges in multi-layer ecosystems deployed in distributed systems: 1) dealing with inconsistency among distributed logs, 2) distinguishing useful information from noisy logging, and 3) probabilistic determination of root causes. These highlighted the gaps in the current approaches and lead to our third contribution.

3. Introduced a new two-phase error diagnosis general framework for distributed software ecosystem from the operator (rather than the developer) perspective. This new approach attempts to remove some inconsistency and noise by combining phase-one local diagnosis with phase-two global diagnosis and produces a probability-ranked list of potential root causes. This simplifies the complexities of constructing correlations between logging information and root causes.

2. Related Works

In previous work, efforts have been placed into the improvement of logging mechanisms for providing more comprehensive system information to assist system management. For example, Apache Flume [2] aims to offer a scalable service for efficiently collecting, aggregating, and moving large amounts of log data in large-scale distributed computing environments. Similar logging systems include Facebook Scribe [9], Netflix Edda [13] and Chukwa [16], which are systems for aggregating real-time streams of log data from a large number of servers. These developments of logging systems provide a good basis for collecting up-to-date system information in complex distributed systems, but they do not have the capability to bridge the gap between logging information and error diagnosis.

Another focus of research of using logging information to assist troubleshooting is to explore effective machine learning approaches for mining critical messages associated with known problems. For example, Xu et. al. [21] studied the correlation between logs and source code. In [12], Nagaraj et. al. troubleshoot performance problems by using machine learning to compare system logging behaviors to

infer associations between components and performance. In [11], Narasimhan and her team members studied the correlation of OS metrics for failure detection in distributed systems. In [24][25][26], Zhou's research group studied the trace of logging information in source codes, and introduced a new logging mechanism to locate the position of bugs with more efficiency. And in [15], Oliner et. al. studied the connections between heterogeneous logs and quantified the interaction between components using these logs. There is a general lack of ecosystem awareness in these tools and the ability to deal with log inconsistency and uncertainty as well as cross system incompatibility.

Misconfigurations are another significant issues leading to software system errors. Zhou and her colleagues conducted an empirical study over different types of misconfigurations and their effects on systems by studying several open source projects, including MySQL, Tomcat and etc. [23]. They focus on the misconfigurations of each individual system, while the correlation of configurations across systems, especially in a distributed environment, is ignored. Randy Katz and his colleagues [17] studied the connection between configuration and software source code to improve misconfiguration detection but did not cover the connection between configurations and logs, which is critical to operators.

These existing works give a good basis for understanding some challenges in error diagnosis. But many studies are from the viewpoint of software developers rather than operators. They also did not consider issues around the connections among the logs and configurations at different layers and across different nodes.

3. Case Study: HBase Cluster on Amazon EC2

Our case study comes from a real world privacy research project where the goal is to process large amounts of anonymised information using different approaches to see if one can still infer identity from the information. Several sub-projects want to share a HBase/Hadoop cluster which is deployed in Amazon EC2. The operators and users of the cluster are IT-savvy researchers and system admins but not Hadoop or distributed system experts. Although Amazon provides an Elastic Map Reduce (EMR) system with Hadoop pre-installed, the different requirements of the sub-projects led to a fresh deployment on EC2 virtual machines.

An HBase/Hadoop cluster consists of Hadoop Distributed File System (HDFS) for distributed files storage, Zookeeper for distributed service coordination, and HBase for fast individual record lookups and updates in distributed files. Each node in an HBase cluster consists of multiple layers of software systems, shown as Figure 1 (a). Every layer must perform in a correct manner to ensure the communication across layers/nodes and overall system availability, as shown in Figure 1 (b).

The communication between nodes in a Hadoop ecosystem relies on SSH connections, so security, ports and protocols required by SSH must be available. Hadoop, Zookeeper and HBase rely on Java SDK. Updated versions of Java that are compatible are necessary. The Hadoop layer is the basis of an HBase cluster. This layer is controlled by HDFS and MapReduce [3]. The configurations over the Namenode and all Datanodes [3] must be correct, ensuring the communication and computation over this layer, so that clients of Hadoop can access HDFS or MapReduce services. (HBase does not need MapReduce, but applications of HBase may require MapReduce). Zookeeper performs a role of distributed service coordinator for HBase. Its responsibilities include tracking server failures and network partitions. Without Zookeeper, HBase is not operational. Based on these underlying distributed services, HBase requires communication between the HMaster and the Regional Servers [5] in the HBase layer. The full deployment and running of some of our small programs went through several false starts in a matter of weeks by different people independently. We asked the people to record their major errors, diagnosis experiences and root causes.

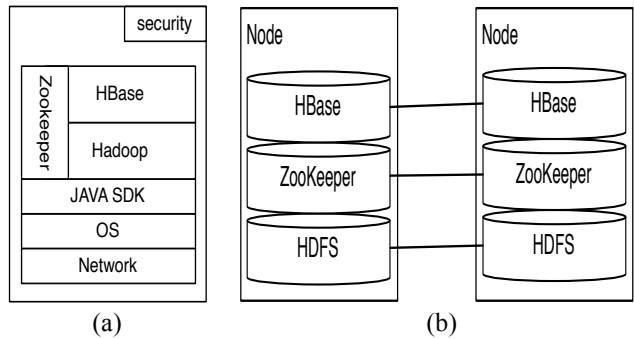


Figure 1 Layers of software systems in Hadoop

4. Logging Exceptions and Uncertainties in Determining Root Causes

In Table 1, we list some key examples of logs and error messages collected in our Hadoop/HBase deployment process. The “Logging Exception” column records the error messages when the deployment process got interrupted. The “Possible Causes” column listed the possible causes and the relevant information that different operators mentally considered or physically examined during error diagnosis. For errors that are related to connection issues, we use Src and Dest to respectively represent the source and destination nodes.

Table 1: Logging Exceptions and Potential Causes

	Source	Logging Exception	Possible Causes: Required Information for Examination
1	HBase/Hadoop	“org.apache.hadoop.hdfs.server.datanode.DataNode: DataNode is shutting down: org.apache.hadoop.ipc.RemoteException: org.apache.hadoop.hdfs.protocol.UnregisteredDataNodeException”	In the problematic DataNodes: <ul style="list-style-type: none"> • Instance is down: <i>ping, ssh connection</i> • Access permission: <i>check authentication keys, check ssh connection</i> • HDFS configuration: <i>conf/slaves</i> • HDFS missing components: <i>check the datanode setting and directories in hdfs</i>
2	Zookeeper	“java.net.UnknownHostException at org.apache.zookeeper.ZooKeeper.<init>(ZooKeeper.java:445)”	In Src and Dest nodes: <ul style="list-style-type: none"> • DSN: <i>DSN configuration and testing</i> • Network connection: <i>ssh testing</i> • Zookeeper connection: <i>JPS and logging messages in zoo.out</i> • Zookeeper configuration: <i>zoo.cfg</i> • Zookeeper status: <i>processes (PID and JPS)</i> • Cross-node configuration consistency
3	HDFS/MapReduce / HBase/ Zookeeper	“java.net.ConnectException: Connection refused “	In Src and Dest: <ul style="list-style-type: none"> • Network connection: <i>ping IPs, ping hostnames and check ssh connection</i> • Security setting: <i>check ssh connection and check authentication keys</i> • Hostname/IP/Ports configuration: <i>check configuration files, netstat and lsof</i> • Software status: <i>check processes</i> • Software compatibility: <i>detect and check system and library versions</i> • Cross-layer configuration consistency • Cross-node configuration consistency
4	HBase/Hadoop	“org.apache.hadoop.hdfs.server.namenode.NameNode: java.lang.IllegalArgumentException: Does not contain a valid host:port authority: file”	In Src and Dest: <ul style="list-style-type: none"> • Missing configuration files: <i>hostfile, hadoop configurations</i> • Security file missing or incorrect: <i>connection permission, host/port permission</i> • Host and Port setting in HDFS: <i>core-site.xml, hdfs-site.xml</i> • Host and Port settings in DNS • Network host and port settings: <i>netstat, lsof etc</i> • Cross-node configuration consistency
5	HBase/Hadoop	“org.apache.hadoop.hdfs.server.common.InconsistentFSStateException: Directory	In the problematic nodes: <ul style="list-style-type: none"> • Missing files in HDFS file system: <i>look for directory in hdfs</i>

		/app/hadoop/tmp/dfs/name is in an inconsistent state: storage directory does not exist or is not accessible.”	<ul style="list-style-type: none"> • Missing/Incorrect operations on HDFS: <i>hdfs format</i> • Directory misconfiguration: <i>core-site.xml</i>
6	HBase/Hadoop	“WARN org.apache.hadoop.metrics2.impl.MetricsSystemImpl: Source name ugi already exists! ERROR org.apache.hadoop.hdfs.server.datanode.Datanode: java.io.IOException: Incompatible namespaceIDs in /app/hadoop/tmp/dfs/data:”	In the problematic NameNode and DataNode: <ul style="list-style-type: none"> • Misconfigurations on the hadoop: <i>scan the name space setting in hadoop</i> • File System duplication: <i>scan the hdfs file system</i> • Other nodes with the same name started: <i>scan configurations and hostfiles</i>
7	Zookeeper	“JMX enabled by default Using config: /home/ubuntu/zookeeper-3.4.5/bin/./conf/zoo.cfg Error contacting service. It is probably not running.”	In the problematic Nodes: <ul style="list-style-type: none"> • Misconfigurations on JAVA: <i>Java version and Java Path</i> • Missing components in JAVA: <i>Update Java version</i> • JAVA configurations in Zookeeper: <i>JAVA_HOME Path</i> • Zookeeper configurations: <i>configurations in zoo.cfg</i> • Zookeeper version problem: <i>the compatibility of Zookeeper, JAVA and OS</i>
8	Hadoop/MapReduce	In deployment testing, “class is not found: maxtempteremapper , and the job is not defined in the jar , when running map reduce jobs...”	In the problematic Nodes: <ul style="list-style-type: none"> • Misconfiguration in Jobtracker: <i>the path to the MapReduce Jar</i> • Misconfigurations in MapReduce: <i>mapred-site.xml</i> • Class compiling by JAVA: <i>the Java compiler</i> • The correctness of the Jar file: <i>the source code of the MR application</i>
9	HBase/Hadoop	“ERROR org.apache.hadoop.security.UserGroupInformation: PriviledgedActionException as:ubuntu cause:java.io.IOException: File /app/hadoop/tmp/mapred/system/jobtracker.info could only be replicated to 0 nodes, instead of 1”	In the problematic Nodes: <ul style="list-style-type: none"> • Security setting: <i>RSA settings</i> • Directory configuration: <i>scan configuration files core-site.xml</i> • HDFS files system directories: <i>scan the Hadoop file system, run hadoop scripts, or scan hadoop log</i>
10	HBase/Hadoop	“FATAL org.apache.hadoop.hdfs.StateChange: BLOCK* NameSystem.getDatnode ... ERROR org.apache.hadoop.security.UserGroupInformation: PriviledgedActionException as:ubuntu cause:org.apache.hadoop.hdfs.protocol.Unregistere dDatanodeException”	In the problematic Nodes: <ul style="list-style-type: none"> • Hadoop misconfiguration: <i>scan hadoop configuration files core-site.xml and conf/slaves</i> • HDFS not formatted: <i>scan hadoop file system, run hadoop scripts</i> • HBase Configurations: <i>scan HBase configurations conf/hbase-site.xml</i> • Cross-layer configuration consistency: <i>scan the configurations with dependencies in HBase and Hadoop</i> • System security: <i>test SSH connctions</i>
11	HBase/Hadoop	“org.apache.hadoop.hbase.client.RetriesExhausted Exception: Failed setting up proxy interface	In the Src and Dest Nodes: <ul style="list-style-type: none"> • Hadoop status: <i>scan processes by PID and JPS, use Hadoop commands</i> • Hadoop client and server configurations: <i>the master name setting in hdfs</i> • Permission in the system: <i>RSA and ssh connctions</i> • Cross-layer configuration consistency: <i>HBase configurations is inconsistent to the Hadoop configuraitons, e.g., the ports and the names of file systems</i>
12	HBass/Hadoop	“WARN org.apache.hadoop.hdfs.server.datanode.Datanode: java.io.IOException: Too many open files at java.io.UnixFileSystem.createFileExclusively(Native Method) at java.io.File.createNewFile(File.java:883) ...	In nodes used by HBase <ul style="list-style-type: none"> • configuration of HBase: <i>maximum number of files setting</i> • Workload of HBase: <i>under heavy work load</i> • Configuration of Hadoop: <i>maximum number of files setting</i> • OS environment misconfiguration: <i>e.g. default ulimit (user file limit) on most unix systems insufficient</i>
13	Hadoop	“org.apache.hadoop.hdfs.DFSClient: DataStreamer Exception: org.apache.hadoop.ipc.RemoteException: java.io.IOException: File /app/hadoop/tmp/mapred/system/jobtracker.info could only be replicated to 0 nodes, instead of 3”	In Src and Dest Nodes <ul style="list-style-type: none"> • Hadoop Status: <i>scan processes by PID and JPS, use Hadoop commands</i> • MapReduce Status: <i>scan processes by PID and JPS, use MapReduce commands</i> • Directory in Hadoop configurations: <i>the number of replicas in hdfs-site.xml, the number of slaves in conf/slaves</i> • Connection problems: <i>e.g. node IP configurations</i> • HDFS file system: <i>the directory does not exist in the HDFS</i> • Cross-node configuration consistency: <i>the Hadoop states in each node</i>
14	Zookeeper	“org.apache.zookeeper.ClientCnxn: Session 0x23d41f532090005 for server null, unexpected error, closing socket connection and attempting reconnect”	In Src and Dest Nodes <ul style="list-style-type: none"> • Zookeeper Configurations: <i>the clinet port, name of nodes etc. in zoo.cfg</i> • Network Configurations: <i>the ssh connections to other nodes</i> • Security Configurations: <i>the RSA settings</i> • Cross-node configuration consistency: <i>the zookeeper configurations in each node, the configuration over networks in each node</i> • States of Zookeeper: <i>running, waiting or failed</i>
15	HBase/Hadoop/Zookeeper	“FATAL org.apache.hadoop.hbase.regionserver.HRegionServer: ABORTING region server hbaseSlave1.60020.1362958856599: Unexpected exception during initialization, aborting org.apache.zookeeper KeeperException\$ConnectionLossException: KeeperErrorCode =	In Src and Dest Nodes <ul style="list-style-type: none"> • HBase configurations: <i>the zookeeper setting in HBase, conf/hbase-site and conf/hbase-env.sh, the authority to use Zookeeper from HBase</i> • The OS/Network problem on the nodes: <i>the ssh connection and the compatibility between JAVA, HBase and OS</i> • Zookeeper configurations: <i>the Zookeeper availability</i> • Cross-layer configuration consistency: <i>the ports, quorum and authority setup</i>

	ConnectionLoss for /hbase/master at org.apache.zookeeper.KeeperException.create(KeeperException.java:99)”	in zookeeper and HBase
--	---	------------------------

From the operator experiences in the project, locating a root cause from a logging exception is very difficult. A logging exception could result from multiple causes while the connections to these causes are not obvious from an error message. For example, a logging “java.net.ConnectException: Connection refused”, shown in Figure 2, has at least 10 possible causes. And exceptions on different software (in the ecosystem) or on different nodes are sometimes inconsistent but related in a direct and indirect manner. It is an extremely exhausting search process to locate a root cause in a large-scale domain with highly coupled information and many uncertainties.

In this study, we classify the error analysis into three layers: exception, source and cause. Exception is the error message returned in log files or console; source is defined as the component that originally leads to this exception message; and cause is the reason that the source got the exception. And we classify errors into four groups: operations, configurations, software and resources. We use these classifications in our proposed approach to organize local diagnosis and a global diagnosis.

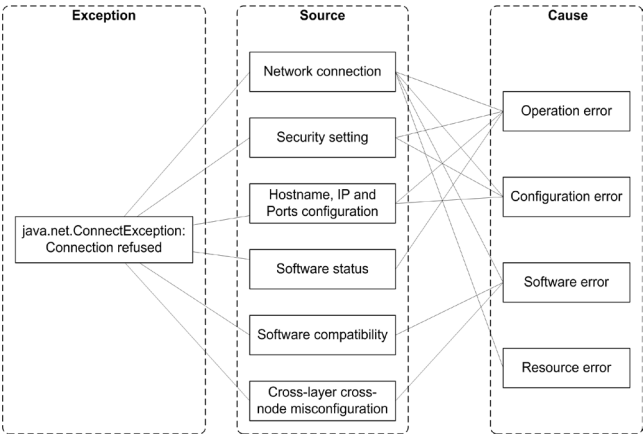


Figure 2 Three layers of error diagnosis: exception-source-cause

Configuration errors

Misconfigurations include legal ones with unintended effects and illegal ones (e.g. lexical, and syntax errors) that are commonly seen in standalone software systems [23]. We also include the cross-domain inconsistent configurations in such distributed ecosystems. The later one is more difficult to detect because all configurations must be taken as a whole for error examination. We give an example that caused issues in the project.

Example 1. HDFS directory used in HBase must be consistent with the Hadoop file system default name. In HBase, hbase-site.xml, the setting of hbase.rootdir:

```
<property>
  <name>hbase.rootdir</name>
  <value>hdfs://hbaseMaster:54310/hbase</value>
</property>
```

must be consistent with the setting of fs.default.name in Hadoop core-site.xml

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://hbaseMaster: 54310/value>
</property>
```

Mismatch of these configurations results in failures of HBase startup. For an enterprise HBase cluster deployment, such as CDH4, there are hundreds of options requiring customizable configurations in 20+ sub-systems [7][17]. These configurations are inter-correlated, but misconfigurations are hard to detect.

Operation errors:

Operation errors include missing operations and incorrect operations. Operation errors cause missing components and abnormal system behaviors, resulting in software failures. For example, HDFS initialization requires a newly formatted file system. Inconsistent File System State Exception shown below will return if this required operation was missing. The formatting is performed externally. The message is not obviously interpretable to lack of formatting.

Example 2:

```
org.apache.hadoop.hdfs.server.common.InconsistentFSStateException: Directory /app/hadoop/tmp/dfs/name is in an inconsistent state: storage directory does not exist or is not accessible.
```

Software errors

Software errors came from software incompatibility and bugs. One instance is the incompatibility between Hadoop 0.20.x version and HBase 0.90.2, resulting in potential data loss [14]. Another commonly seen failure due to system incompatibility is certain required Java libraries do not exist. Such case usually happens because of the incompatibility between Java and the OS, and so some required Java libraries are not installed. Here are two examples of logging errors returned by Hadoop and Zookeeper installation in the project. However, both messages are not at all clear about the root causes and can lead operators to the wrong places. But after examining related logs in other layers, the root cause was located.

Example 3: JAVA problem in Hadoop:

Error msg: "java[13417:1203] Unable to load realm info from SCDynamicStore" when running any HDFS command

Example 4: JAVA problem in ZooKeeper:

JMX enabled by default
Using config: /home/ubuntu/zookeeper3.4.5/bin/../conf/zoo.cfg
Error contacting service. It is probably not running.

Resource errors

Resource errors refer to resource unavailability occurring in the computing environment. For example, limitation of disk I/O (or failure of SAN disks) could result in significant performance degradation in some nodes, resulting in some exceptions of *timeout*. However, one key challenge is that many such resource errors are hidden in log files and not correlated with respective resource metrics. Only by looking at different logs from different layers of software in the ecosystem, can the root cause be identified.

5. Discussion: Three Challenges to Troubleshoot Errors with Logs

Logs guide error diagnosis. There are three challenges that should be addressed for achieving more accurate and efficient error diagnosis in distributed ecosystem.

5.1 Dealing with inconsistency among logs

Inconsistent loggings around states and events introduce significant issues to error diagnosis. Inconsistency may occur in a single log file, across multiple log files in different components. Inconsistency of logging information includes two types: inconsistent contexts and inconsistent timestamps.

Taking a Hadoop ecosystem as an example, an ecosystem consists of a large number of interacting heterogeneous components. Each component has logging mechanism for capturing specific states and events, what messages are put into log files is often determined by the requirements of component itself with no global coordinator for managing these logging messages across components. The decisions of what states and events are put into the log file under what context are not the same in different components. When taking these logging messages across components as a whole for error diagnosis, missing, redundant and contradictory information may introduce context inconsistency.

Another type of inconsistency comes from inconsistent timestamps in large-scale systems where network latency

cannot be ignored. Information logging could be asynchronous as errors and other corresponding information are written into log files. This asynchronous logging contributes to risks of timing inconsistency, which may be misleading in error diagnosis and omit correlated events. Solutions to timing correlation problems exist such as NTP¹ and Google Spanner [8] but these solutions are not currently implemented in our test stack. Again, we are attempting to deal with what is, rather than what should be.

5.2 Distinguishing useful information from noisy logging

Large-scale distributed systems are constantly producing a huge amount of logs for both developers and operators. Collecting all of them into a central system is often itself a significant challenges. Systems have emerged to create such centralized log collection, for example Scribe from Facebook, Flume from Apache, Logstash² and Chukwa [16].

Due to the large amount of information available, error diagnosis is often very time-consuming whether it is done by humans querying the centralized log system or through machine learning systems across all the logs. Traditional error analysis algorithms could encounter scalability issues dealing with a large number of logging messages. Some scalable clusters for logging analysis were developed for addressing this issue [21][22]. But these solutions focus on offline analysis to identify source code bugs while operation issues often require online or nearline analysis putting significant challenge to the analysis infrastructure and algorithm. Thus, it is important to discard noise earlier and effectively for different types of errors at different times.

In many cases, such as performance issues and connection problems, additional tests and associated logs are required for analysis. They are often time consuming if planned and done reactively through human operators. These additional tests should be incorporated into the error diagnosis tools and logging infrastructure so they are automatically carried out at certain stage of the error diagnosis or proactively done, adding more useful signals to the error diagnosis process.

5.3 Probabilistic determination of root causes dealing with uncertain correlations

In error diagnosis, correlation of logging events is critical for identifying the root causes. Many machine-learning techniques have been developed for exploring the correlated events in log files in order to construct more accurate and more comprehensive models for

¹ http://en.wikipedia.org/wiki/Network_Time_Protocol

² <http://logstash.net/>

troubleshooting [11]. However, uncertainties in logs introduce significant challenges in determining root causes. Uncertainties in log files are often caused by missing logging messages, inconsistent information and ambiguity of logging language (lexical and syntax). We classify the uncertainties into four types:

Uncertainties Between Exceptions

In distributed systems, an error occurring in one place often triggers a sequence of responses across a number of connected components. These responses may or may not introduce further exceptions at different components. However, simply mining exception messages from these distributed log files may not detect the connections among these exceptions. Known communications between components should be considered in correlating exceptions and comparing different root causes diagnosis at each component or node.

Uncertainties Between Component States

Accurate logging states and context help filter useless information and guides error diagnosis. They are important information for understanding component statuses and limiting the scope for searching the root cause to errors. Logging states could be fully coupled or fully independent, or with somehow indirect connections. But these dependent relationships among state logging are not described in log files. And missing and inconsistent states logging may further introduce uncertainties in the relationships between states. Dependencies in an ecosystem must be taken into consideration when analysing state logs.

Uncertainties Between Events

In error diagnosis exploring the coherence of logging events is a critical task for tracking the change of system subject to errors, providing a basis for inferring the root cause from exceptions. A challenge for constructing event coherence is uncertainties lying in the relationships between logging events. These uncertainties destroy connections between information, losing data for modeling the sequence of system change subject to errors.

Uncertainties Between States And Events

In most cases, logging states and events must be considered at the same time for modeling the system behavior in terms of logging conditions. Ideally logging messages deliver details of events and of corresponding states across this process. But this obviously is over optimistic. In most log files the connections between states and events contain uncertainties, which destroy the event-state mapping, creating a gap for finding the root causes from logging errors.

6. A Two-Phase Error Diagnosis Framework

The above challenges are the consequence of current logging mechanisms and overall designs, which are often

out of the control of the users. So error diagnosis requires an effective approach that is capable of figuring out the most possible root causes for errors despite of the inconsistency, noise and uncertainty in logs. To achieve this goal in a large-scale distributed computing environment, we are working on two ideas. The first idea is to treat the operations as a set of explicit processes interacting with each other. We model and analyze these processes and track the their progression at runtime. We use the processes to connect seemingly independent events and states scattered in various logs and introduce “process context” for error diagnosis [27]. In this paper, we introduce the second idea, which proposes a two-phase error diagnosis framework for error diagnosis. The first-phase error diagnosis is conducted at each distributed node with agents for local troubleshooting, and a second-phase is performed on a centralized server for global error diagnosis to compare the various local diagnoses and deal with node-to-node errors. Unlike existing solutions that have a centralized database aggregating all logging information, in our approach information is highly filtered for the second-phase diagnosis depending on the error types, environment and local diagnosis.

A framework of this design is shown in Figure 3. The key is to let each node or log-file propose a set of potential causes for the errors (if there are logging exceptions in the file) and gather the states of the relevant components, then send these likely causes and component states to a centralized second-phase diagnosis for probability-ranked list of causes using a gossip algorithm [19]. The logging information that we consider in this framework includes log files from software components, e.g. Hadoop, Zookeeper and HBase, and historical information of resource components, which include records of resource (CPU/Memory) consumption, disk I/O, network throughput, and process states monitored by agent-based systems (e.g. JMX and Nagios in our environment). All of these are seen as log files of components in our approach.

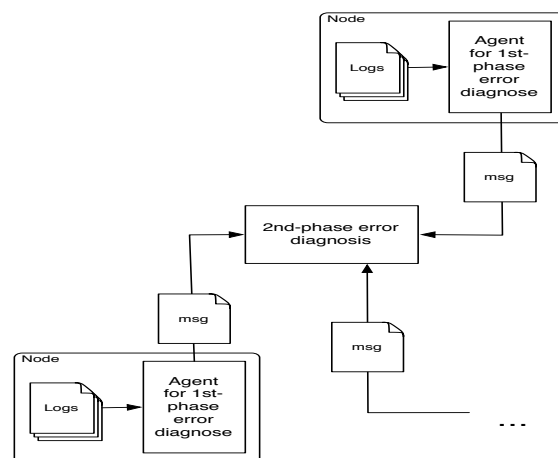


Figure 3: Architecture of the 2-phase error diagnosis

6.1 The first-phase error diagnosis

The first-phase error diagnosis is conducted with agents located at each distributed node for identifying the errors in the components in the node. This process is described with Figure 4.

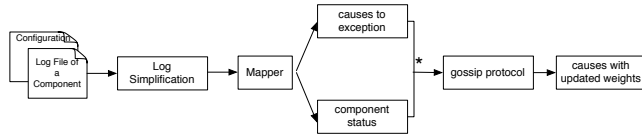


Figure 4: The process of error diagnosis in the first-phase

Inputs to an agent include log files of components and configuration files. An agent first summarizes each log file, which is a process to convert logging information into a standard format with consistent terms (lexical and syntax) for later identification. This operation is conducted in the stage of log simplification. For each summarized log file given by the log simplification, the agent uses a mapper, which is a small expert knowledge base responsible to deliver a set of likely causes in response to the logging exception. A mapper offers: a) a list of candidate causes that may contribute to the logging exceptions (which include ERROR and WARNING messages), denoted by C_e^r , standing for a cause r that may lead to an exception e in component C . Each cause may include a set of sub-causes $C_e^{r'}$, and b) the status of the component, denoted by C_s , which includes the status of domain name, ports, accounts, security, tractable actions for software components, and utilization and performance for resource components. Each C_e^r is associated with a weight w , whose initial value is 1. We define a tuple $[C_e^r, w]$ to indicate this relationship. These proposed causes and monitored component statuses are considered as a whole by a gossip algorithm, updating the weight w of each C_e^r with a rule: when a cause C_e^r conflicts to a component status C_s , the associate weight w is reduced by 1; and when a cause C_e^r is supported by another log file, the weight w is then increased by 1. This strategy reduces the number of correlated features (across logging messages) that are less related to errors, creating potential for handling complex problems in a large-scale systems.

6.1.1 An Example

The following is an example for troubleshooting cross-system inconsistent configuration within an HBase cluster. Cross-system misconfiguration is hard to detect because it is difficult to trace exceptions across multiple systems. In an HBase node (with IP: 10.141.133.22, which is a master node in this cluster), it includes log files respectively from HBase, Hadoop, Zookeeper. When a log file from HBase returns an exception, shown as:

```
2013-03-08 09:31:44,934 INFO org.apache.hadoop.ipc.Client:
Retrying connect to server: hbaseMaster/10.141.133.22:9000.
```

```
Already tried 9 time(s).
```

```
2013-03-08 09:31:44,938 FATAL
```

```
org.apache.hadoop.hbase.master.HMaster: Unhandled exception.
Starting shutdown.
```

```
java.net.ConnectException: Call to
hbaseMaster/10.141.133.22:9000 failed on connection exception:
java.net.ConnectException: Connection refused
```

```
...
```

```
ERROR org.apache.hadoop.hbase.master.HMasterCommandLine:
Failed to start master
```

,where hbaseMaster is a domain name defined in the configuration file. In the phase-one error diagnosis, this logging information is summarized as:

```
HBaseMaster:9000 failed
```

```
connection exception: hbaseMaster/10.141.133.22:9000
```

The mapper takes this as input and returns a set of likely causes to the exception and gives the states of the component:

1. Hadoop server: unavailable ($C_{e_{HD}^{avail}}: 1$)
2. Hadoop server: not accessible ($C_{e_{HD}^{access}}: 1$), whose prerequisite: $C_{s_{HD}^{avail}} = true$, and with sub causes:
 - a. domain name is unavailable ($C_{e_{HD}^{hbaseMaster:9000}}: 1$)
 - b. user account is unavailable ($C_{e_{HD}^{act}}: 1$)
 - c. security setting is unavailable ($C_{e_{HD}^{sec}}: 1$)

```
status: HBase HMaster: failed ( $C_{s_{HBaseHMaster}^{avail}} = false$ ).
```

In the same node, the log file from Hadoop NameNode gives the information

```
2013-03-08 09:23:02,362 INFO
```

```
org.apache.hadoop.hdfs.server.namenode.FSNamesystem: Roll
FSImage from 10.141.133.22
```

```
2013-03-08 09:23:02,362 INFO
```

```
org.apache.hadoop.hdfs.server.namenode.FSNamesystem:
```

```
Number of transactions: 0 Total time for transactions(ms):
```

```
0Number of transactions batched in Syncs: 0 Number of syncs: 1
SyncTimes(ms): 8
```

Because there are no logging exceptions, no causes are proposed from this Hadoop log file. So it can give: $C_{s_{HD}^{avail}} = true$. And since no configurations regarding account and security are found in the configuration files, it gives $C_{s_{HD}^{act}} = true$ and $C_{s_{HD}^{sec}} = true$. And it can be achieved from the summary of Hadoop log that: $C_{s_{HD}^{hbaseMaster:54310}} = true$, where $hbaseMaster:54310$ is the domain name of Hadoop.

A combination of this information given by Mappers is used in the Gossip protocol for updating the weight associated with each proposed cause. Output is shown as below. The reasons are described in the bracket in the right-hand side.

1. $[C_{e_{HD}^{avail}}: 0] (C_{e_{HD}^{avail}} = false \text{ conflicts } C_{s_{HD}^{avail}} = true)$
2. $[C_{e_{HD}^{access}}: 1] (C_{s_{HD}^{avail}} = true, \text{ and no information directly related to } C_{e_{HD}^{access}})$
 - a. $[C_{e_{HD}^{hbaseMaster:9000}}: 1]$ (no information directly related to

$$C_{e_HD}^{hbaseMaster:9000})$$

b. $[C_{e_HD}^{act}:0] (C_{e_HD}^{act}=false \text{ conflicts } C_{s_HD}^{act}=true)$
c. $[C_{e_HD}^{sec}:0] (C_{e_HD}^{sec}=false \text{ conflicts } C_{s_HD}^{sec}=true)$

The cause to this “java.net.ConnectException” is limited to the availability of domain name of *hbaseMaster:9000* (cause 2.a). Although this approach does not provide a 100% accurate error diagnosis, it shows the possibility of using limited information to sort out the most likely causes for a logging error in a complex computing environment with many connected systems.

6.2 The second-phase error diagnosis

The second-phase error diagnosis offers troubleshooting for the exceptions that may be across multiple nodes. This process sorts out the possibility of causes that are delivered by the agents in the first-phase error diagnosis.

Each agent summaries the output of the first-phase error diagnosis into a message, which includes the likely causes with updated weights (if the weight is greater than zero), and the status of each component.

6.2.1 An Example

For example, the agent in the above node will deliver the second-phases error diagnosis a message with the information of:

Agent ID: 10.141.133.22
HBase Log:
 $[C_{e_HD}^{access}:1]$
 $[C_{e_HD}^c:1]$
 $C_{s_HBaseHMaster}^{avail} = false, C_{s_HBaseHMaster}^{act} = true, C_{s_HBaseHMaster}^{sec} = true$
Hadoop Log:
 $C_{s_HD}^{avail} = true, C_{s_HD}^{hbaseMaster:54310} = true, C_{s_HD}^{act} = true, C_{s_HD}^{sec} = true$
Zookeeper Log:
 $[C_{e_ZK}^{hbaseSlave3:3888}:1]$
 $C_{s_ZK}^{avail} = true, C_{s_ZK}^{myID:1} = follower, C_{s_ZK}^{act} = true, C_{s_ZK}^{sec} = true$

This message includes the information of Zookeeper. Because there is a WARN message given in the Zookeeper log file, shown as:

Cannot open channel to 4 at election address hbaseSlave3/10.151.97.82:3888

and the Zookeeper status has shown that this Zookeeper quorum is performing follower, a possible cause with weight is shown as $[C_{e_ZK}^{hbaseSlave3:3888}:1]$. This warning error message is related to another Zookeeper quorum on: *hbaseSlave3:3888*. It is handled by the second-phase error diagnosis.

For this troubleshooting, input information regarding this error for the second-phase error diagnosis includes:

Agent ID: 10.141.133.22, propose error $[C_{e_ZK}^{hbaseSlave3:3888}:1]$

Agent ID: 10.36.33.18, where the zookeeper quorum is selected as leader, propose error $[C_{e_ZK}^{hbaseSlave3:3888}:1]$
Agent ID: 10.151.97.82, where locate the problematic zookeeper quorum *hbaseSlave3:3888*

Because the Zookeeper status is found in the Agent ID: 10.151.97.82, the weight of $C_{e_ZK}^{hbaseSlave3:3888}$ is updated to

$[C_{e_ZK}^{hbaseSlave3:3888}:2]$

in the second-phase error diagnosis with the gossip protocol to find out the most likely cause to guide troubleshooting. It locates the issue on the zookeeper quorum on 10.151.97.82. And since no states of this Zookeeper quorum are returned, the focus of troubleshooting can be limited on:

Network communication between nodes, and
Configurations of the zookeeper quorum in Zookeeper and HBase

This simple example shows that the 2-phase error diagnosis can use existing limited information to determine a list of ranked possible causes to logging errors dealing with uncertainty challenges we identified earlier. And the strategy is simple to implement as it uses an existing gossip algorithm to compare local diagnosis, which could be in turn based on past work and ad-hoc knowledge database, and it can handle cross-layer and cross-node errors.

7. Conclusions and Future Works

Using a real world case study, we identified some difficult-to-diagnosis errors committed by non-expert Hadoop/HBase users. We classified errors and documented the difficulties in error diagnosis, which led to three key challenges in ecosystem error diagnosis. We proposed a simple and scalable two-phased error diagnosis framework that only communicates the absolute necessary information for global diagnosis after local diagnosis. We experimented and demonstrated the feasibility of the approach using a small set of common Hadoop ecosystem errors. We are currently implementing the full framework and performing large-scale experiments.

8. Acknowledgement

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

9. References

- [1] AWS EC2, <http://aws.amazon.com/ec2/>
- [2] Apache Flume, <http://flume.apache.org/> 2013
- [3] Apache Hadoop, <http://hadoop.apache.org/> 2013
- [4] Apache Hadoop, “HDFS High Availability Using the Quorum Journal Manager”, 2013, <http://hadoop.apache.org/docs/r2.0.3-alpha/hadoop->

- yarn/hadoop-yarn-site/HDFSHighAvailabilityWithQJM.html
- [5] Apache HBase, <http://hbase.apache.org/> 2013
 - [6] Apache Zookeeper, <http://zookeeper.apache.org/> 2013
 - [7] Cloudera “CDH4 Installation Guide” 2013
<http://www.cloudera.com/content/cloudera-content/cloudera-docs/CDH4/latest/PDF/CDH4-Installation-Guide.pdf>
 - [8] Corbett, J.C., et al. Spanner: Google’s Globally-Distributed Database, Processings OSDI ’12, Tenth Symposium on Operating System Design and Implementation, Hollywood, Ca, October, 2012.
 - [9] Facebook Scribe, <https://github.com/facebook/scribe>
 - [10] Lars George, “HBase: The Definitive Guide”, Publisher: O’Reilly Media, 2011
 - [11] Soila P. Kavulya, Kaustubh Joshi, Felicita Di Giandomenico, Priya Narasimhan, “Failure Diagnosis of Complex Systems”, In *Journal of Resilience Assessment and Evaluation of Computing Systems* 2012, pp 239-261
 - [12] Karthik Nagaraj, Charles Killian, and Jennifer Neville. “Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems”. In the *Proceedings of 9th USENIX Symposium on Networked Systems Design and Implementation* (NSDI ’12). San Jose, CA. 25-27 April, 2012.
 - [13] Netflix Edda, <https://github.com/Netflix/edda>
 - [14] Michael G. Noll, “Building an Hadoop 0.20.x Version for HBase 0.90.2”, 2011, <http://www.michael-noll.com/blog/2011/04/14/building-an-hadoop-0-20-x-version-for-hbase-0-90-2/>
 - [15] Adam J. Oliner, Ashutosh V. Kulkarni, Alex Aiken. “Using correlated surprise to infer shared influence”, In the *Proceedings of Dependable Systems and Networks* (DSN), 2010, June 28 2010-July 1 2010,
 - [16] Ariel Rabkin, Randy Katz “Chukwa: a system for reliable large-scale log collection”, in *Proceedings of the 24th international conference on Large installation system administration* (LISA 10), 2010
 - [17] Ariel Rabkin, “Using Program Analysis to Reduce Misconfiguration in Open Source Systems Software”, Ph.D thesis 2012
 - [18] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. “Pip: Detecting the Unexpected in Distributed Systems”. In *proceedings of Networked Systems Design and Implementation* (NSDI 2006). May 2006
 - [19] Devavrat Shah, “Gossip Algorithm”, MIT, 2009, <http://web.mit.edu/devavrat/www/GossipBook.pdf>
 - [20] Tom White, “Hadoop: The Definitive Guide”, the second edition, published by O’Reilly Media 2010
 - [21] Wei Xu, “System Problem Detection by Mining Console Logs”, Ph.D thesis, EECS, UC Berkeley, Aug. 2010
 - [22] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan, “Large-scale system problem detection by mining console logs”, In *Proceeding of the 22nd ACM Symposium on Operating Systems Principles* (SOSP’ 09), Big Sky, MT, October 2009
 - [23] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram and Shankar Pasupathy. “An Empirical Study on Configuration Errors in Commercial and Open Source Systems.” In the *Proceedings Of The 23rd ACM Symposium On Operating Systems Principles* (SOSP’11), October 2011
 - [24] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. “Characterising Logging Practices in Open-Source Software”. In the *Proceedings of the 34th International Conference on Software Engineering* (ICSE’12), Zurich, Switzerland, June 2012
 - [25] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou and Stefan Savage. “Improving Software Diagnosability via Log Enhancement”. In *ACM Transactions on Computer Systems* (TOCS), Vol. 30, No. 1, Article 4, February 2012.
 - [26] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou and Stefan Savage. “Be Conservative: Enhancing Failure Diagnosis with Proactive Logging” In the *Proceedings of the 9th ACM/USENIX Symposium on Operating Systems Design and Implementation* (OSDI’12), Hollywood, CA,
 - [27] X. Xu, L. Zhu, J. Li, L. Bass, Q. Lu, and M. Fu, "Modeling and Analysing Operation Processes for Dependability," in *IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN), Fast Abstract, 2013

Installation of an External Lustre Filesystem using Cray esMS management and Lustre 1.8.6

Patrick Webb
webb@cray.com

May 1, 2013

Abstract

High performance computing systems need a similarly large scale storage system in order to manage the massive quantities of data that are produced. The unique aspects of each customer's site means that the on-site configuration and creation of the filesystem will be unique. In this paper we will look at the installation of multiple separate Lustre 1.8.6 filesystems attached to the Los Alamos National Laboratory ACES systems and their management back-end. We will examine the structure of the filesystem and the choices made during the installation and configuration as well the obstacles that we encountered along the way and the methods used to overcome them.

1. Introduction

Every high performance computing system requires an equally high performance filesystem in order to properly manage the massive quantities of data that is produced by the computations ongoing on the machine. The physical installation of our system was performed by trained Cray hardware engineers. The unique challenges of our installation arose with the software portion of the installation. Software is usually the domain of the on-site system analyst team to install and customize to their needs, and in this case Cray has permanent on-site system analysts as part of that team providing the software expertise to install, test, configure and operate the filesystem software.

The installation is designed to be built as an externally connected filesystem that is mounted by the Cielo supercomputer[1], a Cray XE6 system operated by the Los Alamos National Labs, and one of their major HPC resources. Lustre was chosen as a solution due to the experience that Cray has with integrating Lustre into their computational environment, as well being able to provide extensive support for the filesystem.

Lustre is a parallel distributed filesystem, consisting of a series of metadata servers (MDS) which keep track of metadata objects, storage servers (OSS) which manage data storage objects, and object storage targets (OST) which physically store the data objects, arranged in a hierarchical format to allow the distribution of data across many devices. Clients first contact the MDS to begin their transaction, then communicate directly with the appropriate OSS nodes to read/write to an OST. The installed filesystem is connected to the mainframe via an LNet (Lustre Networking) network protocol which provides the communication infrastructure.

The system uses specialized LNet router nodes to translate traffic between the Cray Gemini network (the proprietary Cray interconnect) and Infiniband using the LNet protocol.

In this paper we will explore the methods used to install, test, configure and operate three Lustre 1.8.6 filesystems from the perspective of the permanent Cray on-site system analyst. The filesystems discussed consists of two 2PB systems, one 4PB system, and two 350TB testbed systems. The PB filesystems are attached via fibre-channel to 12, 12 and 24 racks of disk arrays respectively, configured in a RAID6 8+2 format. Management is by a single Dell rack-mount server providing boot images and configuration management to the filesystem nodes. The focus will remain on the Cielo portion of the installation, since many of the unique challenges we encountered manifested within Cielo's environment and scale.

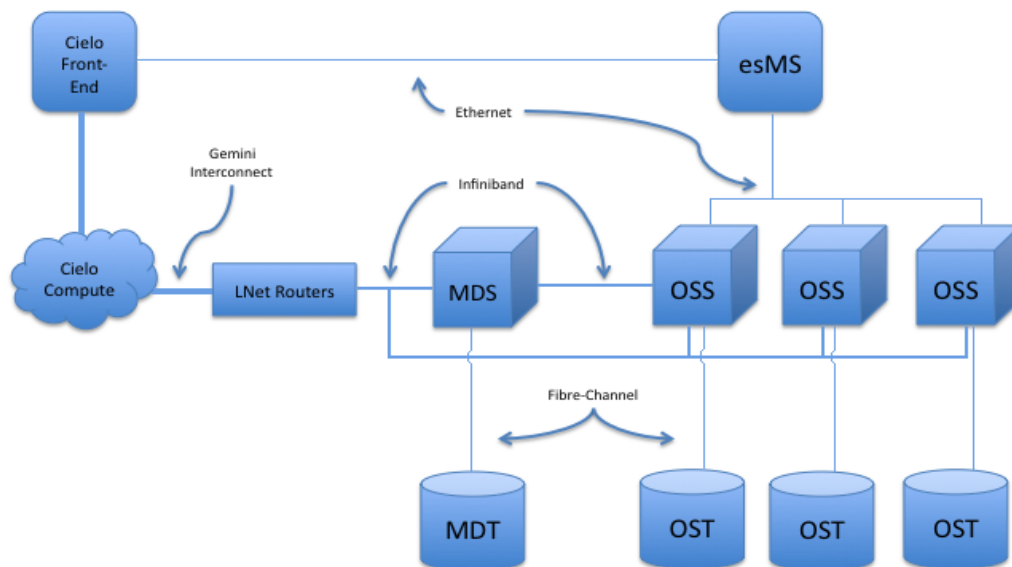


Fig. 1 A simplified diagram of Cielo's Lustre filesystem

2. System Capabilities & Overview

The Cielo Lustre filesystem (dubbed an esFS or external service filesystem in Cray parlance) is a 96 OSS, 6 MDS system connected to 48 storage racks with a total storage capacity of 8PB, managed by a single external service management server (esMS). All of the blades (OSS, MDS and esMS) nodes are Dell R710 blades. The storage racks consist of 128 2TB hard drives apiece configured into an 8+2 RAID controlled by a redundant LSI controller. The network routing on the Cray system side is handled by 104 service nodes configured as LNet routers. The interconnect between the storage racks and the Lustre servers is a fibre channel connection, and between the Lustre servers and the Cielo system is an Infiniband network. The

Infiniband network on Cielo makes use of two Director class Infiniband switches to manage the network. The management network between the esMS and the OSS nodes consists of basic 1GigE Ethernet.

The sum of resources are then split into three different filesystems managed by the single esMS blade: two 2PB filesystems and one 4PB filesystem. Each of the two 2PB filesystems are assigned 12 racks of disks, and the 4PB filesystem is assigned 24 racks.

The Infiniband network is shared between all three filesystems, and connects the Lustre components to the compute portion via an LNet network managed by the Cray LNet router nodes. The LNet routers are pooled together and shared by all three filesystems instead of separating them into smaller groups.

The software stack consists of three separate portions. On the Cielo side, the LNet routers use the Cray Linux Environment (CLE) OS customized with the necessary LNet and Lustre kernel modules. The esMS uses a SLES11 base OS. The OSS and MDS nodes are managed using Bright Cluster Manager (BCM) software running on the esMS. BCM is used to manage the different boot images and configuration options for the OSS and MDS nodes, which PXE boot their OS. The OSS and MDS nodes run a CentOS 5.4 base system customized by Cray with Lustre 1.8.6 software.

The performance of the filesystem is measured across several dimensions, and is described in detail in section 3.4.

3. Challenges

3.1 Initial Setup Challenges

The setup of the esFS system would be the responsibility of the Cray on-site system engineers and system analysts to install, test, and operate the filesystems. The first challenges manifested at the Cray factory where the initial test and development systems would be constructed and configured before shipment. These test systems would be the template for the larger Lustre filesystems, as well as platforms for test and development. One of the site analysts travelled to the Cray factory in order to participate in the construction and learn directly from the engineers assembling the system.

The following elements were constructed at the factory for the test and development system: OSS & MDS hardware configuration, Infiniband network, fiber connections to disk racks, esMS hardware configuration, LUN (a type of storage object) creation, esMS software stack, and the OSS & MDS software stack. The actual Lustre filesystem was not created, and the LNet network that connects the Cray compute hardware to the Lustre filesystem was also not assembled at the factory. The security stance of the LANL site is such that it requires incoming systems to be built up from bare metal, meaning that any assembly at the Cray factory would be useful only for testing purposes. Thus it was critical for the on-site system analysts to learn as much as possible from the Cray engineers. The task of building the entire filesystem and its management node (the esMS) from the ground up would be their responsibility.

3.2 Site Set-up Challenges

The first steps in bringing up the Lustre filesystems was to first build and configure the esMS node which would provision and monitor the OSS and MDS nodes. Despite the fact that the project was on schedule, there was significant pressure to stand up the filesystem as quickly as possible and to not deviate from the setup of the test & development system. However, there was a key critical difference between the test & development system and the full-scale production filesystem: the full-scale system was meant to have a backup esMS node with automatic failover configured. The test and development system had no such esMS backup system configured. The consequence was that the full-scale system was initially configured with only a single esMS node instead of the intended (and required by contract) primary/secondary esMS configuration. Cray documentation for adding a secondary esMS to an already configured and running single esMS didn't exist. We would be the first site to execute this task.

Building a single esMS was a straightforward procedure. It uses the SLES11 operating system as its basis, modified to add Cray Lustre control packages. BCM uses its own installation tool that requires inputting necessary configuration options (network, etc.) and allowing it to set up the entire OS under BCM management. Custom Cray scripts for monitoring and managing automatic failover were also installed at this time.

Once the esMS was fully built and configured it was time to power on and set up the OSS and MDS nodes. During power-up each of the physical nodes were checked in order to confirm that the BIOS settings had been set properly at the factory. A small number of nodes had been overlooked and needed to be reconfigured on-site. Finally, the MDS/OSS node boot images were configured into BCM.

3.3 Configuration Challenges

We decided that we would use the configuration from another Cray installation site, the National Energy Research Scientific Computing (NERSC) Center, as the basis of our own configuration. This met with a few obstacles from a managerial perspective. The desire to have as safe and stable system as possible meant that there was a great deal of pushback against any sort of deviation from a known quantity, namely the NERSC configuration. However, we faced a few issues that made duplicating NERSC unreasonable. First, the scale of the LANL filesystem was much larger than NERSC. Second, the LNet and Infiniband network at LANL used a very different set of hardware. Finally the software stack at LANL, unlike NERSC, was productized into a cohesive package managed by BCM.

3.4 Testing & Acceptance Challenges

The testing plan for the Lustre filesystem measured the baseline hardware performance, the ability to meet a minimum level of filesystem performance, and the ability of the system to ride through an interruption of one or more of the hardware components. Each Infiniband link between MDS/OSS nodes and LNet nodes were tested at ~2.7GB/s average per link. Aggregated, the system saw a

maximum raw throughput of ~70.3GB/s between 52 LNet and 48 OSS nodes. Under load, the system saw a peak of 77.4GB/s for a 2k core job (65.5GB/s required). Metadata operations showed ~22k-24k creates/11k-18k deletes per second (10k/s each required) when each core operated on its own file. All performance tests passed with only minor adjustments to meet requirements.

The fault injection tests tested for events such as a power failure, node crash, or network failure. The deliverables required automatic component failover and stated that the system would be able to automatically failover an ailing component in the following circumstances: A normal shutdown of an LSI Controller, MDS, or OSS node; An unexpected power failure of an LSI Controller, MDS, or OSS node; A loss of an LNet router; The loss of network connectivity between the Infiniband switch and an MDS, OSS, or LNet router; Loss of one or both fibre channel connection between an OSS node and an LSI controller. Of these tests, all had to either continue to serve data albeit at a degraded performance, or signal an IO error that would unambiguously indicate that IO was the fault of the job failing.

Tested failures degraded performance during recovery from no measurable impact (LNet router failure) to as much as 87% of peak, and/or caused an acceptable IO error (OSS, LSI Controller, etc.). Lustre attempts to rescue transactions from the failed components, and transactions that don't recover are discarded to avoid storing corrupted data. After recovery, performance degrades roughly proportional to the amount of filesystem resources made unavailable.

Despite these requirements, the monitoring and failover scripts were released to the customer capable only of automatically failing over a node if network connectivity was lost, or if the node panic'd and froze but remained powered on.

The orderly shutdowns of the various hardware components were not designed to initiate a failover on the assumption that if an orderly shutdown were taking place, that the responsible administrator would have either quiesced the system or manually instigated a failover in order to power off a node. A node simply being "off" meant that the monitoring system would not know if it had already performed a failover (A failing node is "STONITHed", or powered off, in order to ensure that it will not interfere with its backup.) or if that node had freshly failed. Erring towards safety, the monitoring software would not initiate a failover for a node that was simply turned off. This behavior also affected how the system responded to an unexpected power loss, namely that it did not initiate a failover.

Other fault injection tests were never designed to initiate an automatic failover, or even interrupt operations of the filesystem. The LSI controllers used a shared power supply that was internally redundant and powered pairs of controllers, so a power loss would always affect both, but never a single controller. Fibre-channel connections were not designed to be monitored by the esMS or the OSS/MDS nodes, and their redundant connection meant that losing one connection meant there were still routes available to connect to the disk racks. The fault injection testing proved as much, with minimal impact on performance.

The LNet network had another set of challenges that only arose at scale. The LNet network check that ran on each of the OSS and MDS nodes would ping a randomly chosen peer somewhere out on the Infiniband network, and if that ping were successful it would report back that it had passed. If that ping timed out, then

it would report a failure and the esMS would initiate a failover. Internally, BCM executes these checks serially every few minutes. At scale, we found ourselves monitoring 96 nodes spread across three different filesystems. The check executed every 60s, but it took as much as 90s for a failed node to report that its ping had timed out and failed. Due to the serial nature of BCM's testing, this meant that if a node near the end of the list of nodes to check were to fail, the timeout for the ping (and thus the affirmative "failed" condition) would not complete and notify the esMS. The esMS assumes a 'pass' if not explicitly notified that a node had failed, and would have already moved on to the next iteration of checks and discarded the results of the previous pass. We needed to change the behavior of the monitoring scripts dramatically.

The solutions to our mismatched expectations of our monitoring and failover scripts are described in section 5 below. It caught the management team off guard, and required close collaboration between the developers and field personnel to effect a solution in the field.

3.5 Operational Challenges

Few operational challenges arose. The stability of the filesystem was such that its popularity among the users rose to the point of the system beginning to show signs of strain due to heavy load. Despite users doing their utmost to eke every last bit of performance out of the filesystem, it remained, and still remains, incredibly stable.

Once the system was up and tested and released to users, we began to see a series of false-positive events triggered by the network checks in our monitoring scripts. The first check to throw out false-positives and cause unintended automatic failovers was the LNet network connectivity check. We had already tinkered with the timing during the initial fault injection testing to validate the check. Now the check was too sensitive. Lustre uses only one transaction credit allocated to pings, and prioritizes that very low. High traffic on the system meant that a ping could easily end up timing out if its wait in the queue took longer than 90 seconds (the test timeout parameter) to complete. Subsequent LNet pings could and would succeed, but the health check relied on a single ping to initiate a failover event.

Even checks such as TCP ping and power status checks began to see events such as these as the system load increased and the responsiveness of the OSS and MDS nodes became sluggish. Since all of these checks relied on a single ping or poll, it became more and more likely that one of those checks would time out. Without retries of these checks, a healthy yet busy node would be considered unhealthy. Again, the design of our health checks had serious flaws.

4. Resolutions

4.1 Initial Set-up

Education of the site system analysts was critical in this phase in order to ensure that the proper expertise would be on-hand when the system would be built on-site. This was accomplished by sending one of the site analysts to the Cray

factory for a week to shadow the system construction and spend face-to-face time with the developers. By having the site analyst in the factory, that analyst was also able to get hands-on experience with building up the filesystem while having the Cray development team on hand to instruct them through the process. Valuable to the developers was the ability to closely watch how an admin who had not been involved in the design of the system would follow the installation documentation, and thus improve the quality of the documentation.

4.2 Site set-up

Arguably the biggest obstacle during the set-up was the installation of the backup esMS. Lacking Cray documentation, the admins performing the installation found themselves in a difficult position. The solution was to bring in direct assistance from the developers to bypass and fix issues in the procedure that prevented moving forward. Little troubleshooting was needed, as this was fresh ground. The process involved repartitioning an in-use disk to make partitions that would be mounted by the backup esMS, then migrating data to the new partitions. Next, the backup esMS would mount those portions and make an initial copy. From there, the backup esMS would monitor the primary for failure, and make periodic incremental updates from the primary. The process of adding the backup esMS highlighted many weaknesses in the documentation and initial setup configuration that needed clarification and correction, and instigated the improvements to the Cray documentation. Overall, despite the problems it introduced, the delayed inclusion of the backup esMS improved the quality of the entire esFS installation procedure, which can then be shared with other Cray sites.

4.3 Configuration

The NERSC configuration served as an excellent starting point for the initial setup and configuration. The main resolution to this particular point of the installation was to make effective arguments for the necessity of changing the configuration to better match our hardware. The integrated software stack meant that configuration for the OSS and MDS nodes could be managed from a central location. Scaling was larger, so certain parameters in the LNet configuration in terms of numbers of transfer credits and length of timeouts had to be adjusted upwards in order to handle the additional load. Finally, the biggest difference was the configuration of the LNet routers into a single pool shared between all three filesystems rather than dividing them up into separate networks or even down to a fine-grained routing. Pooling the routers has potential loss of performance due to needing to switch traffic, and risks of instabilities if an LNet router fails spectacularly. However, the Director-class Infiniband switches provide plenty of horsepower to allow a pool configuration to work without a performance impact. With a pool of LNet routers, the set-up and configuration was much simpler (simply place them all into the same network), and it provided a great deal of redundancy in that if any LNet router failed, the traffic that router was serving could easily be sent through any other router on the network.

4.4 Testing & Acceptance

The Cray development team quickly provided an updated rpm that enabled failover for the contractually required failover triggers. The scripts were in fact already capable of performing failover actions in all required cases, but the tests simply had not yet included the code to initiate those actions. The updated RPM simply empowered those tests to carry out failovers.

In-field rewrites of the monitoring and failover scripts were the solution to the problem of LNet network checks not completing and bypassing themselves. We first monitored the return values from the nodes. Noting that nodes at the end of the node list weren't reporting back before a new health check started we then compared timing values. Noting the mismatch between LNet ping timeout, we then wrote into the check script a progressive timeout logic that checked to see if the test passed immediately, within 5 seconds, 10 seconds, etc. until ultimately the test failed and a failure was reported. The code sped up the checks on a healthy system, and left plenty of time available for a failed check to fully timeout. The modifications were fed back to the development team, who integrated them into the code base. However, the new code did not yet address the issue of an otherwise healthy but heavily loaded system from failing a single lnet ping check when a re-try would confirm that the lnet network is perfectly functional.

Poorly understood fault injection tests, namely the LSI controller tests, were solved through frank and earnest discussions the engineers and management staff. The previously existing trust between the two parties made it easy to explain the technical realities, and agree on the necessary reinterpretation of the results. All the people working were fully invested in putting forth their very best work.

4.5 Operations

Once again, in-field changes to the monitoring scripts were necessary to check the status of the networks without failing due to a mere single TCP ping, or LNet ping, timing out. We were able to discover the false positives examining internal Lustre stats, and discovering that the system would periodically oversubscribe its available credits, including the ping credit. The Cray development team took a proactive approach, and added into the code base retries for all appropriate health checks. The system analysts implemented a field fix of disabling active failure in favor of notifying via pager the analysts in the event of specific health checks failing. They kept field fixes in place while waiting for the next polished version of the esFS monitoring scripts were released.

5. Lessons Learned

Recognize and react to the differences between the test and production systems. – The difficulty of adding the backup esMS after the full installation was a troublesome and dangerous procedure that was forced by prioritizing the deadline and slavishly sticking to mirroring the test & development system. If the production and test systems will differ by design, prepare for the installation plan between the two to differ as well.

Documentation of the underlying structure is incredibly valuable. – Knowledge of the underlying structure of the various parts of the esMS/esFS systems was critical to solving many of the build problems, namely the esMS backup.

Embrace the fact that your installation will be unique. – A great deal of discomfort was felt over the fact that the actual configuration parameters differed from the model. Realizing that we must differ smoothed out the decision making and allowed for more rational choices in configuration.

Test all of the contractual requirements as early as possible. – We came very close to having real problems with contractual obligations in our failover scripts. While we were able to add in the required aspects, had we tested them earlier there would have been less pain involved.

Empower the local site analysts to create and implement fixes in the field. – The fact that the local analysts were not only able, but encouraged to implement their own fixes led to quick and effective solutions. It gave the site analysts a sense of ownership of the system, and gave the developers a short-cut to improving the overall code base.

6. Conclusions

The installation of a new filesystem is a complex task with many moving parts, that was only complicated by the fact that many tasks that could have been performed and tested in a factory setting were required to be completed in the field. In addition, the entire product was one of the first releases of the actual productization of the Cray esFS filesystem. The challenges of building such a large installation were met with a great deal of dedication and expertise on the part of the developers and site system analysts. The expected challenges of configuring the different aspects of the network, formatting the filesystem, installing the management software, testing performance, etc. were all present and expediently dealt with.

We were able to respond to the various unexpected challenges with in-field fixes that were later integrated into the release products and made available for other sites to use. Additionally, we were able to keep to the timetable due to the proactive nature of the implementation of these fixes in the field rather than waiting on a development cycle to provide a patch. This kind of dynamic relationship with the home office based developers proved to be an exceptionally strong one that produced effective solutions very quickly.

The final result of this work is an exceptionally stable and popular filesystem that has exceeded the users expectations for availability, stability, and performance. While improvements can always be made, the efforts made during the initial set up will, in my opinion, pay off in terms of the long-term health of the filesystem.

References:

- [1] – C. Lueninghoener *et al.*, “Bringing Up Cielo: Experiences with a Cray XE6 System”, in Proceedings Large Installation System Administration Conf., 2011.